Software Fault Identification via Dynamic Analysis and Machine Learning

by

Yuriy Brun

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

Author ....
Department of Electrical Engineering and Computer Science
August 16, 2003

Certified by..
Michael D. Ernst
Assistant Professor
Thesis Supervisor

Accepted by ..... [illegible] ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Software Fault Identification via Dynamic Analysis and Machine Learning

by

Yuriy Brun

Submitted to the Department of Electrical Engineering and Computer Science
on August 16, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

I propose a technique that identifies program properties that may indicate errors. The technique generates machine learning models of run-time program properties known to expose faults, and applies these models to program properties of user-written code to classify and rank properties that may lead the user to errors.

I evaluate an implementation of the technique, the Fault Invariant Classifier, that demonstrates the efficacy of the error finding 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. Given a set of properties produced by the program analysis, some of which are indicative of errors, the technique selects a subset of properties that are most likely to reveal an error. The experimental evaluation over 941,000 lines of code, showed that a user must examine only the 2.2 highest-ranked properties for C programs and 1.7 for Java programs to find a fault-revealing property. The technique increases the relevance (the concentration of properties that reveal errors) by a factor of 50 on average for C programs, and 4.8 for Java programs.

Thesis Supervisor: Michael D. Ernst
Title: Assistant Professor

Acknowledgments

I would like to thank my advisor, Michael Ernst, for all the patience, guidance, and leadership he has brought to this project. He has provided invaluable experience and support throughout my work with the Program Analysis Group. As far as advisors go, and I've had a few over the years, he has proven himself as a reliable and caring mentor who takes extraordinary amounts of time out of his schedule to promote the wellbeing of his students.

A great deal of progress on my research was made possible by the contributions of fellow researchers, especially Ben Morse and Stephen McCamant. Ben and Stephen provided help in modifying source subject programs used in the experimental evaluation. Thank you to David Saff, who provided the FDanalysis subject programs used in the experimental evaluation. I also thank Ben, Stephen, David, and all my colleagues in the Program Analysis Group, in particular Alan Donovan, Lee Lin, Jeff Perkins, and Toh Ne Win for their continuing suggestions and contributions to this thesis.

Thank you to Gregg Rothermel, who provided the original sources for the subject programs used in the experimental evaluation.

Thank you to Ryan Rifkin who implemented a support vector machine learner SVMfu and provided feedback and support of the software.

I would also like to thank my parents Tatyana and Yefim Brun and sister Dina Brun for their continuous support and encouragement. The values they have instilled in me proved extremely motivating in my undergraduate, and now graduate careers.

Finally, I would like to thank Kristin Jonker for her friendship and help throughout the many nights turned mornings that allowed this work to proceed.

Contents

List of Figures

Chapter 1

Introduction

Programmers typically use test suites to detect errors in programs. 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 the remaining errors. 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 thesis can lead programmers to latent code errors. The technique does not require a test suite 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 ranking or a subset of those properties such that the highly-ranked properties, or the properties in the reported subset, are more likely than average to indicate faults in the program. The program properties may be generated by an arbitrary program analysis; these experiments use a dynamic analysis, but the technique is equally applicable to static analysis.

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.

This technique helps the programmer find errors in code and is most useful when errors in a program are hard to find. It is common for developers to be aware of thousands of errors in a project, and be unable to fix all the errors because of time and other resource limitations. This technique can be used to detect properties of most critical errors, to help correct those errors first. By training machine learning models on properties of past projects' most critical errors, the technique selects the properties that expose errors most like those ones, letting the programmers correct the most critical errors first. For example, the authors of a new operating system can use past versions of operating systems to create models of fault-revealing properties that have proven costly, such as errors that caused a computer crash and required a reboot, errors that required the company to release and distribute software updates, etc. Thus, if a company wanted to lower the number of software updates it had to release, it could create a model of faulty code of past operating systems that required updates, and find such errors in the new operating system, before releasing it to the users.

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 and non-erroneous programs; it creates a machine learning model of properties that expose errors. (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 likely to indicate errors. A programmer searching for latent errors or trying to increase confidence in a program can focus on those properties.

The experiments demonstrate that the technique's implementation, the Fault Invariant Classifier, is able to recognize properties of errors in code. The *relevance* (also known as *utility*) of a set of properties is the fraction with a given desirable property. The output of the machine learning technique's implementation has average relevance 50 times that of the complete set of properties. Without use of the tool, the programmer would have to examine program properties at random or based on intuition.

This thesis argues that machine learning can be used to identify certain program properties that I call fault-revealing. Although the thesis does not give a rigorous proof that these properties lead users to errors in code, it does contain an intuition argument and also sample evidence of such properties from real programs that do lead to errors, in section 6.4.

Chapter 2

Related Work

This research aims to indicate to the user specific program properties that are likely to result from code errors. Because the goal of locating errors is so important, numerous other researchers have taken a similar tack to solving it.

Xie and Engler [21] 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 mine in three respects. First, they use a statically computed metric, whereas I use a dynamically analysis. Second, they increase relevance by 45%–100%, whereas my technique increases relevance by an average of a factor of 49.6 (4860%). Third, their experimental analysis is at the level of an entire source file. By contrast, my technique operates on individual program properties. Rather than demonstrating that a file is more likely to contain an error, my experiments measure whether the specific run-time properties identified by my 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.

Like my 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 any 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 uses three programs, one of which had real faults, and measures number of faulty executions detected rather than number of underlying faults detected. My research identifies suspicious properties rather than suspicious executions, but relies on a similar assumption regarding machine learning being able to make clusters that are dominantly faulty or dominantly correct.

Hangal and Lam [9] 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. My 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, my technique can find latent errors that are present in most or all executions, rather than focusing on anomalies.

Groce and Visser [8] 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 succeeding runs.) These properties abstract away from the specific details of individual counterexamples or successes, freeing users from those tasks. My research also generalizes over successes and failures, but uses a noise-resistant machine learner and applies the resulting models to future runs.

Chapter 3

Technique

This chapter describes the error detection technique that the Fault Invariant Classifier implements. The technique consists of two steps: training and classification. Training is a preprocessing step (Section 3.1) that extracts properties of programs for which errors are known a priori, converts these into a form amenable to machine learning, and applies machine learning to form a model of fault-revealing properties. Classification step is the user suppling the model (Section 3.2) with properties of new code to select the fault-revealing properties, and using those properties to locate latent errors in the new code.

The training step of the technique requires programs with faults and versions of the same programs with those faults removed. The programs with faults removed need not be error-free, and the ones used in the experimental evaluation described in chapter 5 did contain additional errors. (In fact, some additional errors were discovered in other research that used the same subject programs [10].) It is an important feature of the technique that the unknown errors do not hinder the technique; however, the model only captures the errors that are removed between the versions.

3.1 Creating Models

Figure 3-1 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

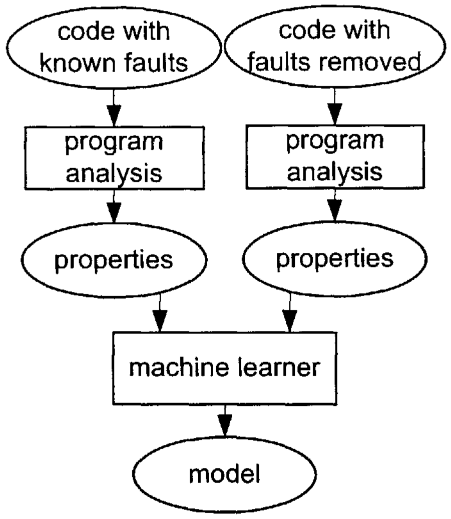


Figure 3-1: Creating a program property model. Rectangles represent tools, and ovals represent tool inputs and outputs. This entire process is automated. The model is used as an input in Figure 3-3. The program analysis is described in Section 4.1, and the machine learner is described in Section 4.3.

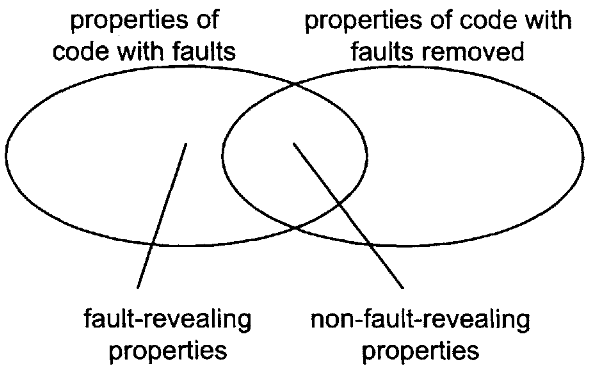

properties of code with faults
properties of code with faults removed
fault-revealing properties
non-fault-revealing properties

Figure 3-2: Fault-revealing program properties are those that appear in code with faults, but not in code without faults. Properties that appear only in non-faulty code are ignored by the machine learning step.

of programs with known errors and corrected versions of those programs. First, program analysis generates properties of programs with faults, and programs with those faults removed. Second, a machine learning algorithm produces a model from these properties. Figure 3-3 shows how the technique uses the model to classify properties.

Before being inputted to the machine learning algorithm, each property is converted to a characteristic vector and is labeled as fault-revealing or non-fault-revealing (or is possibly discarded). Section 4.2 describes the characteristic vectors. Proper-

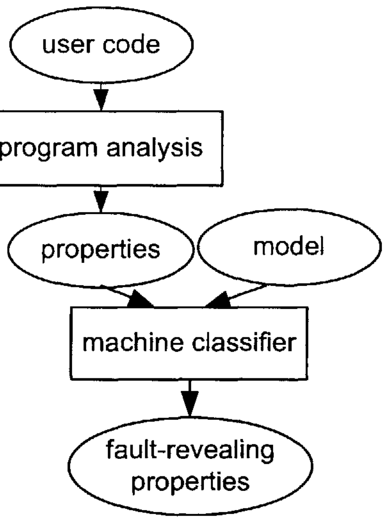


Figure 3-3: Finding likely fault-revealing program properties using a model. Rectangles represent tools, and ovals represent tool inputs and outputs. This entire process is automated. The model is produced by the technique of Figure 3-1.

ties that are present in only 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 3-2).

3.2 Detecting Faults

Figure 3-3 shows how the Fault Invariant Classifier runs on 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.

Chapter 4

Tools

This chapter describes the tools used in the experimental evaluation of the Fault Invariant Classifier technique. 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

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 include using a static analysis (such as abstract interpretation [2]) to obtain semantic properties, or using syntactic properties such as duplicated code [21]. 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.

Daikon, a dynamic invariant detector, generates 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 is syntactically identical to a formal specification, including 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 variable trace that contains, for each execution of a program point, the values of all variables in scope at that point. Each of a set of possible invariants is tested against various combinations of one, two, or three traced variables.

Section A.1 (45) contains a complete list of properties extracted by Daikon. 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 ($\mathsf{x} \in \{\mathsf{a,b,c}\}$), lying in a range ($\mathsf{a} \leq \mathsf{x} \leq \mathsf{b}$), non-zero, modulus ($\mathsf{x} \equiv \mathsf{a} \pmod{\mathsf{b}}$), linear relationships ($\mathsf{z} = \mathsf{ax} + \mathsf{by} + \mathsf{c}$), ordering ($\mathsf{x} \leq \mathsf{y}$), 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, or 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 $\mathsf{p.left} \in \mathsf{mytree}$".

A property is reported only if there is adequate statistical evidence for it. In particular, if there are an inadequate number of observations, observed patterns may be mere coincidence. Consequently, for each detected property, Daikon computes the probability that such a property would appear by chance in a random set of samples. The property is reported only if its probability is smaller than a user-defined confidence parameter [6].

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 3-1 and 3-3.)

A characteristic vector is a sequence of boolean, integral, and floating point values. Each value is placed into its own *slot* in the vector. It can be thought of as a point in multidimensional space.

For example, suppose there were a total of four slots: one to indicate whether the property is an equality, one to indicate whether the property is a $\geq$ relation, one to report the number of variables, and one to indicate whether the property is over floating point values. A property $x = y$, where x and y are of type int, would have the values of 1 (or yes) for the first slot, 0 (or no) for the second slot, 2 for the third slot, and 0 (or no) for the fourth slot. Thus the property $x = y$ would be represented by the vector $\langle 1,\ 0,\ 2,\ 0\rangle$. Likewise, $a \geq b + c$, where a, b, and c are floating point variables, would re represented by the vector $\langle 0,\ 1,\ 3,\ 1\rangle$.

A characteristic vector is intended to capture as much of the information in the property as possible. Overall, the characteristic vectors contain 388 slots. The machine learning algorithms of Section 4.3 are good at ignoring irrelevant slots. Appendix A contains a complete listing of the slots generated by the converter.

Daikon 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]. Other slots represent the type of a property (e.g., two variable equality such as $x = y$, or containment such as $x \in val\_array$).

During the training step only, each characteristic vector is labeled as fault-revealing, labeled as non-fault-revealing, or discarded, as indicated in Figure 3-2.

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 by repeating the smaller set. This normalization is necessary because some machine learners interpret non-equal class sizes as indicating that some misclassifications are more undesirable than others.

4.3 Machine Learning Algorithms

The experiments use two different machine learning algorithms: support vector machines and decision trees. Section 6.3 presents the advantages each machine learner offered to the experiments. Machine learners treat each characteristic vector as a point in multi-dimensional space. 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.

Machine learners generate models during the training step, and provide a mechanism for applying those models to new points in the classification step.

4.3.1 Support Vector Machine Learning Algorithm

A support vector machine (SVM) [1] considers each characteristic vector to be a point in a multi-dimensional space; there are as many dimensions as there are slots in the vector. The learning algorithm accepts labeled points (in these experiments there are exactly two labels: fault-revealing and non-fault-revealing), and it tries to separate the labels via mathematical functions called kernel functions. The support vector machine chooses the instantiation of the kernel function that best separates the labeled points; for example, in the case of a linear kernel function, the SVM selects a plane. Once a model is trained, new points can be classified according to which side of the model function they reside on.

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. 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 implementation of support vector machines, SVMlight [12] and SVMfu [14], 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. They did, however, represent some clusters, so the top ranking properties were often fault-revealing.

The results reported in this paper use the SVMfu implementation [14].

4.3.2 Decision Tree Machine Learning Algorithm

A decision tree (also known as identification tree) machine learner [20] 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.)

A decision tree is equivalent to a set of if-then rules (see section 6.3.2 for an example). Decision trees cannot be used to rank properties, but only to classify them. 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, as opposed to a single kernel function.

Optionally, decision tree learning allows boosting to refine models to cover a larger number of separated clusters of points. Boosting trains an initial model, and then trains more models on the same training data, such that each subsequent model emphasizes the points that would be 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.

The experiments use the C5.0 decision tree implementation [13].

Chapter 5

Experiments

This chapter describes the methods used in the experimental evaluation of the Fault Invariant Classifier.

5.1 Subject Programs

Experimental evaluation of the Fault Invariant Classifier uses twelve subject programs. Eight of these are written in C and four are written in Java. In total, the twelve programs are 624,000 non-comment non-blank lines of code (941,000 with comments and blanks).

5.1.1 C programs

There are eight C programs used as subjects in the experimental evaluation of the technique. Seven of the eight were created by Siemens Research [11], and subsequently modified by Rothermel and Harrold [15]. 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 as realistic errors as possible, that reflected their experience with real programs.

	Average			Faulty	Total	
Program	Functions	NCNB	LOC	versions	NCNB	LOC
print_tokens	18	452	539	7	3164	4313
print_tokens2	19	379	489	10	3790	5373
replace	21	456	507	32	14592	16737
schedule	18	276	397	9	2484	3971
schedule2	16	280	299	10	2800	3291
space	137	9568	9826	34	325312	334084
tcas	9	136	174	41	5576	7319
tot_info	7	334	398	23	7682	9552
Total	245	11881	12629	166	361048	406445

Figure 5-1: C 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 commends and blanks. The print_tokens and print_tokens2 programs are unrelated, as are the schedule and schedule2 programs.

The researchers then discarded faulty versions that failed too few or too many 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, each program follows its own distinct specification and has different sets of legal inputs and outputs.

The eighth 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 [19] and Graves et al. [7]. Figure 5-1 summarizes the size of the programs, as well as the number of faulty versions for each.

5.1.2 Java programs

There are four used as subjects in the experimental evaluation of the technique. Three programs were written by MIT's *Laboratory in Software Engineering* 6.170 class as solutions to class assignments. Each student submits the assignment solution during the term, and then, after getting feedback, gets a chance to correct errors in the code and resubmit the solution. The resubmitted solutions are typically very similar code to the original solutions, but with errors removed.

Program	Average			Faulty versions	Total	
	Functions	NCBC	LOC		NCBC	LOC
Geo	49	825	1923	95	78375	115364
Pathfinder	18	430	910	41	17630	54593
Streets	19	1720	4459	60	103200	267534
FDAnalysis	277	5770	8864	11	63470	97505
Total	363	7145	16156	207	262675	534996

Figure 5-2: Java programs used in the experimental evaluation. NCBC is the number of non-comment non-blank lines of code; LOC is the total number of lines with comments and blanks.

Because the student Java programs may, and often do, contain multiple errors in each version, a larger fraction of the properties are fault-revealing than for the other programs. As a result, there is less room for improvement for the Java programs (e.g., picking a property at random has a 12% chance of picking a fault-revealing one for Java programs, and only a 0.9% chance for C programs).

The fourth Java program, FDAnalysis takes as input a set of test suite executions, and calculates the times at which regression errors were generated and fixed [16]. The FDAnalysis program was written by a single graduate student at MIT, who made and discovered eleven regression errors in the process. 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.

Figure 5-2 summarizes the sizes of the four Java programs.

5.2 Procedure

My evaluation of the Fault Invariant Classifier implementation uses two experiments regarding recognition of fault-revealing properties. The first experiment uses support vector machines as the machine learner and the second experiment uses decision trees. The goal of these experiments is to determine whether a model of fault-revealing properties of some programs can correctly classify the properties of another program. The experiments use the programs described in section 5.1. 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 and measure the accuracy of the classification against the known correct labeling, determined by comparing the properties of the faulty version and the version with the fault removed (Figure 3-2).

Some machine learners are able to output a quality score when classifying properties. The Fault Invariant Classifier uses this quality score to limit the number of properties reported, which makes the technique more usable.

My evaluation consists of two experiments, one to evaluate each of the two machine learners, C5.0 decision trees and SVMfu support vector machines. Each experiment is performed twice, once on the eight C programs, and once on the four Java programs. Each experiment trains a machine learner model based on the fault-revealing properties of all but one program, and then tests the effectiveness of the model on the properties of the last program, the one not included in training.

5.3 Measurements

My 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 [17, 18] is a measure of usefulness of the output, the ratio of the number of correctly identified fault-revealing properties over the total number of properties considered:

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

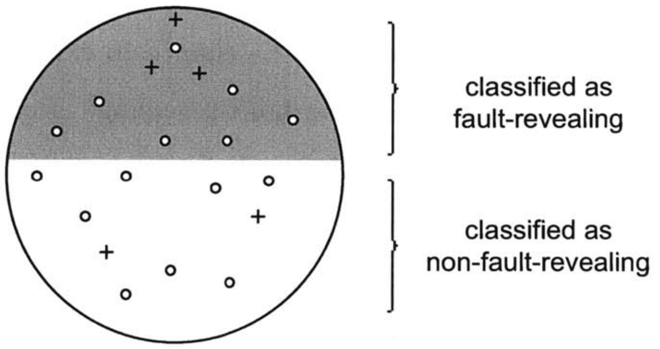
Overall relevance is the relevance of the entire set of all program properties for a given program. *Classification relevance* is the relevance of the set of properties reported as fault-revealing. *Fixed-size relevance* is the relevance of a set of preselected size; I selected 80 because it is the size that maximized average relevance for all the programs in the experiments. Relevance of a set of properties represents the likelihood of a property in that set being fault-revealing. I define the brevity of a set of properties as the inverse of the relevance, or the average number of properties a user must examine to find a fault-revealing one. Brevity is also measured over the overall set of properties, classification set, and fixed-size set.

The importance of the classification relevance measure is that it is the fraction of properties that the tool classifies as fault-revealing. If user wishes to explore all the properties that the tool reports as having a chance to expose an error, the user should expect that fraction of them to be fault-revealing. Fixed-size relevance is that fraction of properties that are fault-revealing in a set of a given size. That is, if the user examines only the top 80 properties, or if the tool is configured to report only 80 properties, the user should expect that fraction of properties to be fault-revealing. Brevity represents the average, or expected number of properties a user has to examine to find a fault-revealing one, and since I believe that users want to examine the minimum number of properties before finding a fault-revealing one, brevity provides insight into the efficiency of the use of the user's time.

For example, suppose a program has 20 properties, 5 of which are fault-revealing. Further, suppose that the technique classifies 10 properties as fault-revealing, 3 correctly and 7 incorrectly, and of the top two ranked properties one is fault-revealing and the other is non-fault-revealing, as shown in Figure 5-3. Figure 5-3 is a 2-dimensional projection of a multidimensional space.

In the example in Figure 5-3, the original relevance is 0.25 and the fixed-size relevance is 0.5. The improvement in relevance is 2 times.

The best achievable brevity is 1, which happens exactly when relevance is 1. A brevity of 1 means all properties are guaranteed to be fault-revealing.


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 5-3: Example of the relevance and brevity measures. Fault-revealing properties are labeled with crosses, non-fault-revealing properties are labeled with circles. 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).

Chapter 6

Results and Discussion

The experimental evaluation, described in this chapter, showed that the Fault Invariant Classifier implementation of the error-finding technique is capable of classifying properties as fault-revealing. For C programs, on average 45% of the top 80 properties are fault-revealing, so the user only has to examine 2.2 properties to find a fault-revealing one. For Java programs, 59% of the top 80 properties are fault-revealing, so the user only has to look at 1.7 properties to find a fault-revealing one.

The results show that ranking and selecting the top properties is more advantageous than selecting all properties considered fault-revealing by the machine learner.

This chapter is organized as follows. Section 6.1 presents the results of the experimental evaluation. Section 6.2 observes a formation of clusters within the ranking of properties. Section 6.3 discusses advantages and disadvantages of using support vector machines and decision trees as the machine learner. Section 6.4 presents some data on sample user experience with the tool. Section 6.5 discusses my findings regarding what makes some properties fault-revealing.

6.1 Results

Figure 6-1 and 6-2 show the data for the experiments that evaluate the technique, with the fixed size of 80 properties. The first experiment uses SVMfu as the machine learner, and the second uses C5.0.

Figure 6-2 shows the data for the experiments with the Java programs. The SVMfu classification relevance differed little from overall relevance; 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. The Java program improvements are smaller because there was more room for improvement in the C programs, as described in section 5.1.2. The C programs averaged 0.009 relevance before application of the technique, while Java programs averaged 0.120 relevance.

Figure 6-3 shows how set size affects relevance. This figure shows the data from the experiment over C programs. The average fixed-size relevance, over all programs, is maximal for a set of size 80 properties. I computed the data for this figure by measuring the relevance of each program version and computing the average for each fixed-size set. Property clusters, described in section 6.2, cause the flat part of the curve on the left side of the graph.

I am greatly encouraged by the fact that the technique performs on the largest program, space, far better than average. This fact suggests that the technique is scalable to large programs.

6.2 Ranked Property Clustering

The results of the experiment that uses SVMfu as the machine learner reveal that properties are classified in clusters. That is, when ordered by rank, properties are likely to appear in small groups of several fault-revealing or non-fault-revealing properties in a row, as opposed to a random distribution of fault-revealing and non-fault-revealing properties. I believe that these clusters form because small groups of program properties are very similar. In other words, some fact about the code is brought forward in more than one property, and if that fact exposes a fault, then all those

	Relevance			
		SVMfu		C5.0
Program	Overall	Class-ification	Fixed-size	Class-ification
print_tokens2	0.012	0.222	0.050	0.012
print_tokens	0.013	0.177	0.267	0.015
replace	0.011	0.038	0.140	0.149
schedule2	0.011	0.095	0.327	0.520
schedule	0.003	0.002	0.193	0.003
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
Average	0.009	0.010	0.446	0.047
Brevity	111	100	2.2	21.3
Improvement	—	1.1	49.6	5.2

Figure 6-1: C program 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. The fixed size is 80 properties (see Figure 6-3). Brevity of a set is the size of an average subset with at least one fault-revealing property, or the inverse of relevance.

	Relevance			
		SVMfu		C5.0
Program	Overall	Class-ification	Fixed-size	Class-ification
Geo	0.120	0.194	0.548	0.333
Pathfinders	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
Average	0.122	0.332	0.586	0.336
Brevity	8.2	3.0	1.7	3.0
Improvement	—	2.7	4.8	2.7

Figure 6-2: Java program 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. The fixed size is 80 properties (see Figure 6-3). Brevity of a set is the size of an average subset with at least one fault-revealing property, or the inverse of relevance.

properties will be fault-revealing. In Figure 6-3, the flat start of the curve is a result of such clustering. For each program versions, the first few program properties are either all fault-revealing, or non-fault-revealing, thus the relevance over all versions is exactly equal to the fraction of versions that have a highest-ranked fault-revealing

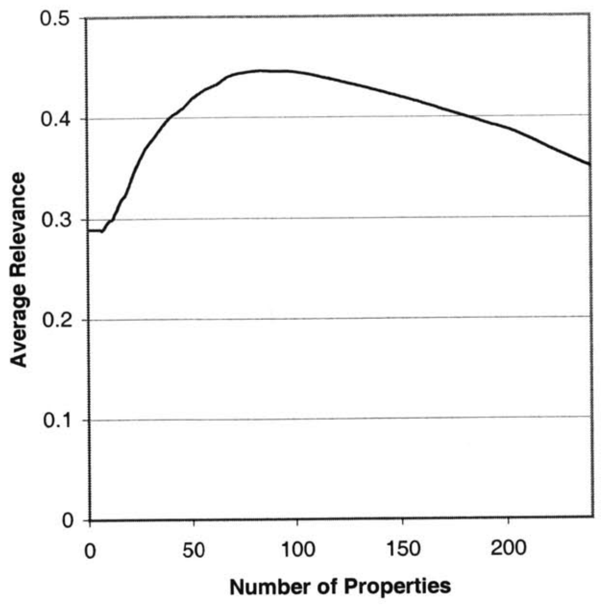


Figure 6-3: Relevance vs. set size averaged across all C program versions using the machine learner SVMfu. Beyond 250 properties, the relevance drops off approximately proportionally to the inverse of the set size. Property clusters, described in section 6.2, cause the flat part of the curve on the left side of the graph.

cluster. After a dozen or so properties, the variation in cluster size makes the curve smoother, as some clusters end and new ones start in other versions, varying the relevance.

The clusters suggest that it may be possible to filter incorrectly identified outlier properties by selecting those that lie in clusters.

6.3 Machine Learning Algorithm Advantages

While decision trees and support vector machines try to solve the same problem, their approaches are quite different. Support vector machines offer some advantages, described in section 6.3.1, while decision trees offer advantages of their own, described in section 6.3.2.

6.3.1 SVM Advantages

Support vector machines are capable of ranking properties. Ranking, as opposed to classification, of properties proved significantly more useful for two reasons. First, the fixed-size relevance was fifty times greater than the overall relevance, while the classification relevance for SVMfu was only marginally better than the overall relevance. Second, the classification set size, or number of properties reported as likely fault-revealing, was too large to expect a user to examine. Building support vector machine models and applying those models to program properties allowed for ranking of the properties which created smaller output sets with higher relevance. The experiments showed that looking at just 2.2 properties of C programs and 1.7 properties of Java programs, on average, is sufficient to find a fault-revealing property. This statistic is important because it indicates the expected work for a user. Examining a smaller number of properties means less work for a user in order to locate an error.

6.3.2 Decision Tree Advantages

Decision tree models were able to improve the classification relevance just like 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 support vector machine models, the rule sets produced by decision trees are easy to read and may provide insights into the reasons why some properties are classified as fault-revealing. 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."

I attempted to use boosting with decision trees, as described in section 4.3.2. In the experiments, boosting had a no significant effect on relevance. I suspect that a nontrivial subset of the training properties misclassified by the original model were outliers, and training additional models while paying special attention to those outliers as well as the other truly fault-revealing properties neither hurt nor improved

Program	Description of fault	Fault-revealing property	Non-fault-revealing property
replace	*maxString* is initialized to 100 but *maxPattern* is 50	$maxPattern \geq 5$	$lin \neq null$
schedule	*prio* is incorrectly set to 2 instead of 1	$(prio \geq 2) \Rightarrow return \leq 0$	*prio_queue* contains no duplicates

Figure 6-4: 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 fault. All four properties were classified correctly by the Fault Invariant Classifier.

the overall models. The resulting models classified more of the properties correctly, as fault-revealing, but at the same time misclassified more outliers.

6.4 User Experience

The experiments indicate that machine learning can identify properties that are likely fault-revealing. Intuition, and the related work in section 2, indicate that fault-revealing properties should lead users to find errors in programs. However, I have not performed a user study to verify this claim. This section provides examples of fault-revealing properties and some non-fault-revealing properties, to give the reader an intuition of how fault-revealing properties can lead users to errors.

Figure 6-4 provides two examples of fault-revealing properties (one for each of two different erroneous programs), and two examples of non-fault-revealing properties for the same two faulty versions. The fault-revealing and non-fault revealing property examples appear in the same methods (addstr for `replace`replace and upgrade_process_prio for `schedule`).

The first example in Figure 6-4 relates to the program `replace`, which is a regular expression search-and-replace routine. The program initialized the maximum input string to be of length 100 but the maximum allowed pattern to only 50. Thus if a user entered a pattern that matched a string correctly, and that pattern was longer than 50 characters, the faulted by treating valid regular expression matches as mismatches. The single difference in the properties of this version and a version with a correctly initialized pattern is that one method addstr in the faulty version was always called when *maxPattern* was greater than or equal to 50.

The second example in Figure 6-4 relates to the program `schedule`, a program that arranges a set of tasks with given priorities. The program's input is a sequence of tasks with priorities, and commands regarding those tasks, e.g., changing priorities. In the faulty version, when the user tried to increase the priority of a job to 1, the software actually set the priority to 2. The fault-revealing property for this program version is that a function returned a non-positive number every time priority was 2 or greater.

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

The fault-revealing properties are supposed to lead programmers to errors in code by attracting the programmers' attention to methods that contain the errors and variables that are involved with the errors. The examples shown in Figure 6-4 provide some evidence that fault-revealing properties do in fact expose methods and variables that reveal errors. To generate more solid evidence to support the claim, one could design a user study where users were asked to remove errors from programs, some with the help of the Fault Invariant Classifier, and other without its help.

6.5 Important Property Slots

One advantage of using decision trees as the machine learner is the human-readability of the models themselves. The models form a set of if-then rules that can be used to explain why certain properties are considered fault-revealing by the machine learning classifier, while other properties are not.

I generated decision tree models based on fault-revealing and non-fault-revealing properties of the programs described in section 5.1, one model per program, and examined those models for common rules. I believe that those common rules are the ones that are most applicable to the general program property.

The following are some if-then rules that appeared most often:

If a property was based on a large number of samples during test suite execution — or in other words, properties of code that executes often — 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.

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 (this rule was present in most models). If the property was also over top level variables (e.g., array x is a top level variable, where as an array which is a field of an object, such as obj.x is not a top level variable), then even more likely (more models included this rule) the property is 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 one or fewer elements, then that property is non-fault-revealing.

Chapter 7

Future Work

In the experiments, the Fault Invariant Classifier technique was accurate at classifying and ranking properties as fault-revealing and non-fault-revealing. The experiments show that the technique's output can be refined to be small enough not to overwhelm the user, and this thesis has begun to argue, by presenting examples, that the fault-revealing properties are useful to locating errors in code. The next logical step is to evaluate the technique in use by programmers on real code errors, by performing a case study to determine whether the tool's output helps users to locate and remove errors.

Programmers can greatly benefit from knowing why certain properties are considered fault-revealing and others are not. Additionally, interpreting decision tree models of fault-revealing properties, as shown in section 6.5, can provide insight into the reasons properties reveal faults and explain why the Fault Invariant Classifier technique works. The knowledge can also indicate how to improve the grammar of the properties and allow for more fault-revealing properties.

The experiments are over a limited set of programs. These programs are widely available and have been used in previous research, permitting comparison of results, and they have multiple realistic faults and test suites. However, they have problems that constitute threats to validity; for example, in addition to size the faults for seven of the eight C programs were injected by the same ten programmers. These programs were appropriate for an initial evaluation of the technique, but a stronger evaluation will execute the experiments on more, larger, and more varied programs. Expanding the program suite will indicate which programs the technique is most effective for and generalize the results reported in this thesis.

A number of future directions are possible regarding the machine learning aspect of this work. For example, one could augment existing machine learning algorithms 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 approach may improve the relevance of the technique because in the current state, some machine learners may not accurately represent multiple clusters within one model.

The clustering idea can extend even further via a detailed analysis of the requirements of the learning problem and development of an expert machine learning algorithm that would specialize in learning program property models. A specialized algorithm may greatly increase the classification power and relevance of the technique.

This thesis has demonstrated the application of the property selection and ranking technique to error location. It may be possible to apply the technique to select properties that improve code understanding or are helpful in automatic proof generation. It may also be used to select properties that expose only a single type of error, e.g., buffer overrun errors or system failure errors.

Chapter 8

Contributions

This thesis presents the design, implementation, and evaluation of an original program analysis technique that uses machine learning to select program properties. 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 expose faults caused by such errors. This thesis also demonstrates the ability of a machine learning algorithm to select program properties based on models of properties known to expose faults. It is a promising result that a machine learner trained on faults in some programs can successfully locate different faults in different programs.

The experimental evaluation of the technique uses an implementation called the Fault Invariant Classifier. The evaluation reports experimental results that quantify the technique's ability to select fault-revealing properties. In the experiments over twelve programs with a total of 624,000 non-blank non-comment lines of code (941,000 with comments and blanks), the 80 top-ranked properties for each program were on average 45% fault-revealing for C programs and 57% for Java programs, a 50-fold and 4.8-fold improvement, respectively, over the fraction of fault-revealing properties in the input set of properties. Further, the technique ranks the properties such that, on average, by examining 2.2 properties for the C programs, and 1.7 properties for the Java programs, the user is likely to encounter at least one fault-revealing property.

I provide some preliminary evidence that links fault-revealing properties to errors in code, and suggest a user study that can be used to provide more solid evidence.

I also present some preliminary analysis of machine learning models that reflect the important aspects of fault-revealing properties, that can help programmers better understand errors.

Appendix A

Definitions of Slots

This appendix contains the definitions of all 388 slots used in the Fault Invariant Classifier implementation of the error-finding technique. Program properties are converted to mathematical vectors by measuring various values of the property and filling the vector's slots with those values. The slots described below are specific to program properties extracted by Daikon. A subset of the slots are all the types of properties that Daikon extracts. These types are listed separately in section A.1. Slots that deal with the location of the property in the code are listed in section A.2, and slots that deal with the properties' variables are listed in section A.3. Finally, section A.4 lists slots that are specific to only certain properties and other general slots.

The Fault Invariant Classifier extracts all slots dynamically. The experimental evaluation relies only on the slots described in this appendix.

A.1 Property Type Slots

Each program property has a type. The slot vector reserves a slot for every type of property; these slots have a value of either 1 or 0, indicating the type of a given property. For any one property, the vector contains a 1 in one of these slots and 0 in all the others. Note that the same properties over different types of variables, e.g., integers or doubles, have different names. The information presented here is also available in the Daikon user manual [3].

- CommonFloatSequence: Represents double sequences that contain a common subset. Prints as "{e1, e2, e3, ...} subset of x[]."

- CommonSequence: Represents long sequences that contain a common subset. Prints as "{e1, e2, e3, ...} subset of x[]."

- CommonStringSequence: Represents string sequences that contain a common subset. Prints as "{s1, s2, s3, ...} subset of x[]."

- DummyInvariant: This is a special property used internally by Daikon to represent properties whose meaning Daikon doesn't understand. The only operation that can by performed on a DummyInvariant is to print it. For instance, dummy invariants can be created to correspond to splitting conditions, when no other property in Daikon's grammar is equivalent to the condition.

- EltLowerBoundFloat: Represents the property that each element of a double[] sequence is greater than or equal to a constant. Prints as "x[] elements $\geq$ c."

- EltNonZero: Represents the property "x $\neq$ 0" where x represents all of the elements of a long sequence. Prints as "x[] elements $\neq$ 0."

- EltNonZeroFloat: Represents the property "x $\neq$ 0" where x represents all of the elements of a double sequence. Prints as "x[] elements $\neq$ 0."

- EltOneOf: Represents long sequences where the elements of the sequence take on only a few distinct values. Prints as either "x[] == c" (when there is only one value), or as "x[] one of {c1, c2, c3}" (when there are multiple values).

- EltOneOfFloat: Represents double sequences where the elements of the sequence take on only a few distinct values. Prints as either "x[] == c" (when there is only one value), or as "x[] one of {c1, c2, c3}" (when there are multiple values).

- EltOneOfString: Represents String sequences where the elements of the sequence take on only a few distinct values. Prints as either "x[] == c" (when there is only one value), or as "x[] one of {c1, c2, c3}" (when there are multiple values).

- EltUpperBound: Represents the property that each element of a long[] sequence is less than or equal to a constant. Prints as "x[] elements $\leq$ c".
- EltUpperBoundFloat: Represents the property that each element of a double[] sequence is less than or equal to a constant. Prints as "x[] elements $\leq$ c."
- EltwiseFloatEqual: Represents equality between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as "x[] elements are equal."
- EltwiseFloatGreaterEqual: Represents the property "$\geq$" between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as "x[] sorted by "$\geq$.""
- EltwiseFloatGreaterThan: Represents the property ">" between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as "x[] sorted by ">.""
- EltwiseFloatLessEqual: Represents the property "$\leq$" between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as "x[] sorted by "$\leq$.""
- EltwiseFloatLessThan: Represents the property "<" between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as "x[] sorted by "<.""
- EltwiseIntEqual: Represents equality between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as "x[] elements are equal."
- EltwiseIntGreaterEqual: Represents the property "$\geq$" between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as "x[] sorted by "$\geq$.""
- EltwiseIntGreaterThan: Represents the property ">" between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as "x[] sorted by ">.""
- EltwiseIntLessEqual: Represents the property "$\leq$" between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as "x[] sorted by "$\leq$.""

- EltwiseIntLessThan: Represents the property "<" between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as "x[] sorted by "<."

  Equality: The Equality property is used for displaying several equality Comparison properties ("x == y", "x == z") as one Equality property ("x == y == z"). This class is created after the actual property detection, and right before printing hence this is not a real property class; it does not implement many of the methods that most property classes do. Furthermore, calling arbitrary methods on this class may not work.

- FloatEqual: Represents a property of "==" between two double scalars.

- FloatGreaterEqual: Represents a property of "≥" between two double scalars.

- FloatGreaterThan: Represents a property of ">" between two double scalars.

- FloatLessEqual: Represents a property of "≤" between two double scalars.

- FloatLessThan: Represents a property of "<" between two double scalars.

- FloatNonEqual: Represents a property of "≠" between two double scalars.

- FunctionBinary: Represents a property between three long scalars by applying a function to two of the scalars. Prints as either "x == function (y, z)" or as "x == y op z" depending upon whether it is an actual function call or a binary operator.

  long operators are: * / % >> << >>> & && ^ | ||
  long functions are: min max gcd pow

- FunctionBinaryFloat: Represents a property between three double scalars by applying a function to two of the scalars. Prints as either "x == function (y, z)" or as "x == y op z" depending upon whether it is an actual function call or a binary operator.

  Current double operators are: /
  Current double functions are: min max pow

- FunctionUnary: Represents a property between two long scalars by applying a function to one of the scalars. Prints as either "x == function(y)" or "x = [op] y" depending upon whether it is an actual function call or a unary operator.

  Current long functions are:

- FunctionUnaryFloat: Represents a property between two double scalars by applying a function to one of the scalars. Prints as either "x == function(y)" or "x = [op] y" depending upon whether it is an actual function call or a unary operator.

  Implication: The Implication property class is used internally within Daikon to handle properties that are only true when certain other conditions are also true (splitting).

- IntEqual: Represents a property of "==" between two long scalars.
- IntGreaterEqual: Represents a property of "$\geq$" between two long scalars.
- IntGreaterThan: Represents a property of ">" between two long scalars.
- IntLessEqual: Represents a property of "$\leq$" between two long scalars.
- IntLessThan: Represents a property of "<" between two long scalars.
- IntNonEqual: Represents a property of "$\neq$" between two long scalars.
- LinearBinary: Represents a Linear property ($y = ax + b$) between two long scalars.
- LinearBinaryFloat: Represents a Linear property ($y = ax + b$) between two double scalars.
- LinearTernary: Represents a Linear property (i.e., $z = ax + by + c$) over three long scalars.
- LinearTernaryFloat: Represents a Linear property (i.e., $z = ax + by + c$) over three double scalars.

- LowerBound: Represents the property 'x $\geq$ c', where c is a constant and x is a long scalar.

- LowerBoundFloat: Represents the property 'x $\geq$ c', where c is a constant and x is a double scalar.

  Member: Represents long scalars that are always members of long sequences. Prints as "x in y[]" where x is a long scalar and y[] is a long sequence.

- MemberFloat: Represents double scalars that are always members of double sequences. Prints as "x in y[]" where x is a double scalar and y[] is a double sequence Modulus Represents the property "x == r (mod m)" where x is a long scalar, r is the remainder, and m is the modulus.

- NoDuplicates: Represents long sequences that contain no duplicate elements. Prints as "x[] contains no duplicates."

- NoDuplicatesFloat: Represents double sequences that contain no duplicate elements. Prints as "x[] contains no duplicates."

- NonModulus: Represents long scalars that are never equal to r (mod m) (for all reasonable values of r and m) but all other numbers in the same range (i.e., all the values that x doesn't take from min(x) to max(x)) are equal to r (mod m). Prints as "x $\neq$ r (mod m)" where r is the remainder and m is the modulus.

- NonZero: Represents long scalars that are non-zero. Prints as either "x $\neq$ 0" or "x $\neq$ null" for pointer types.

- NonZeroFloat: Represents double scalars that are non-zero. Prints as :x $\neq$ 0."

- OneOfFloat: Represents double variables that take on only a few distinct values. Prints as either "x == c" (when there is only one value), or as "x one of {c1, c2, c3}" (when there are multiple values).

- OneOfFloatSequence: Represents double[] variables that take on only a few distinct values. Prints as either "x == c" (when there is only one value), or as "x one of {c1, c2, c3}" (when there are multiple values).

- OneOfScalar: Represents long scalars that take on only a few distinct values. Prints as either "x == c" (when there is only one value), "x one of {c1, c2, c3}" (when there are multiple values), or "x has only one value" (when x is a hashcode (pointer) - this is because the numerical value of the hashcode (pointer) is uninteresting).

- OneOfSequence: Represents long[] variables that take on only a few distinct values. Prints as either "x == c" (when there is only one value), or as "x one of {c1, c2, c3}" (when there are multiple values).

- OneOfString: Represents String variables that take on only a few distinct values. Prints as either "x == c" (when there is only one value), or as "x one of {c1, c2, c3}" (when there are multiple values).

- OneOfStringSequence: Represents String[] variables that take on only a few distinct values. Prints as either "x == c" (when there is only one value), or as "x one of {c1, c2, c3}" (when there are multiple values).

- PairwiseFloatComparison: Represents a property between corresponding elements of two double sequences. The length of the sequences must match for the property to hold. A comparison is made over each x[i], y[i] pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as "x[] [cmp] y[]" where [cmp] is one of $== \neq > \geq < \leq$.

- PairwiseFloatEqual: Represents a property between corresponding elements of two double sequences. The length of the sequences must match for the property to hold. A comparison is made over each x[i], y[i] pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as "x[] == y[]."

- PairwiseFloatGreaterEqual: Represents a property between corresponding elements of two double sequences. The length of the sequences must match for the property to hold. A comparison is made over each x[i], y[i] pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as "x[] $\geq$ y[]."

- PairwiseFloatGreaterThan: Represents a property between corresponding elements of two double sequences. The length of the sequences must match for the property to hold. A comparison is made over each x[i], y[i] pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as "x[] > y[]."

- PairwiseFloatLessEqual: Represents a property between corresponding elements of two double sequences. The length of the sequences must match for the property to hold. A comparison is made over each x[i], y[i] pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as "x[] $\leq$ y[]."

- PairwiseFloatLessThan: Represents a property between corresponding elements of two double sequences. The length of the sequences must match for the property to hold. A comparison is made over each x[i], y[i] pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as "x[] < y[]."

- PairwiseFunctionUnary: Represents a property between corresponding elements of two long sequences by applying a function to one of the elements. The length of the sequences must match for the property to hold. The function is applied to each (x[i], y[i]) pair. Prints as either "x[] == function(y[])" or "x[] = [op] y[]" depending upon whether it is an actual function call or a unary operator.

  Current long Functions are:

- PairwiseFunctionUnaryFloat: Represents a property between corresponding elements of two double sequences by applying a function to one of the elements. The length of the sequences must match for the property to hold. The function is applied to each (x[i], y[i]) pair. Prints as either "x[] == function(y[])" or "x[] = [op] y[]" depending upon whether it is an actual function call or a unary operator.

- PairwiseIntComparison: Represents a property between corresponding elements of two long sequences. The length of the sequences must match for the property to hold. A comparison is made over each x[i], y[i] pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as "x[] [cmp] y[]" where [cmp] is one of $== \neq > \geq < \leq$.

- PairwiseIntEqual: Represents a property between corresponding elements of two long sequences. The length of the sequences must match for the property to hold. A comparison is made over each x[i], y[i] pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as "x[] == y[]."

- PairwiseIntGreaterEqual: Represents a property between corresponding elements of two long sequences. The length of the sequences must match for the property to hold. A comparison is made over each x[i], y[i] pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as "x[] $\geq$ y[]."

- PairwiseIntGreaterThan: Represents a property between corresponding elements of two long sequences. The length of the sequences must match for the property to hold. A comparison is made over each x[i], y[i] pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as "x[] > y[]."

- PairwiseIntLessEqual: Represents a property between corresponding elements of two long sequences. The length of the sequences must match for the property to hold. A comparison is made over each x[i], y[i] pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as "x[] $\leq$ y[]."

- PairwiseIntLessThan: Represents a property between corresponding elements of two long sequences. The length of the sequences must match for the property to hold. A comparison is made over each x[i], y[i] pair. Thus, x[0] is compared to y[0], x[1] to y[1], and so forth. Prints as "x[] < y[]."

- PairwiseLinearBinary: Represents a linear property (i.e., y = ax + b) between the corresponding elements of two long sequences. Each (x[i], y[i]) pair is examined. Thus, x[0] is compared to y[0], x[1] to y[1] and so forth. Prints as "y[] = a * x[] + b."

- PairwiseLinearBinaryFloat: Represents a linear property (i.e., y = ax + b) between the corresponding elements of two double sequences. Each (x[i], y[i]) pair is examined. Thus, x[0] is compared to y[0], x[1] to y[1] and so forth. Prints as "y[] = a * x[] + b."

- Reverse: Represents two long sequences where one is in the reverse order of the other. Prints as "x[] is the reverse of y[]."

- ReverseFloat: Represents two double sequences where one is in the reverse order of the other. Prints as "x[] is the reverse of y[]."

- SeqComparison: Represents properties between two long sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] [cmp] y[] lexically" where [cmp] can be == $<$ $\leq$ $>$ $\geq$. If order doesn't matter for each variable, then the sequences are compared to see if they are set equivalent. Prints as "x[] == y[]." If the axillary information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqComparisonFloat: Represents properties between two double sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] [cmp] y[] lexically" where [cmp] can be == $<$ $\leq$ $>$ $\geq$. If order doesn't matter for each variable, then the sequences are compared to see if they are set equivalent. Prints as "x[] == y[]." If the extra information (e.g., order matters) doesn't match then no comparison is made at all. SeqComparisonString Represents properties between two String sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] [cmp] y[] lexically" where [cmp] can be == $<$ $\leq$ $>$ $\geq$. If order doesn't matter for each variable, then the sequences are compared to see if they are set equivalent. Prints as "x[] == y[]." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqFloatComparison: Represents double scalars with a property to each element of double sequences. Prints as "x[] elements [cmp] y" where x is a double sequence, y is a double scalar, and [cmp] is one of the comparators $== < \leq > \geq$.

- SeqFloatEqual: Represents double scalars with a property to each element of double sequences. Prints as "x[] elements == y" where x is a double sequence and y is a double scalar.

- SeqFloatGreaterEqual: Represents double scalars with a property to each element of double sequences. Prints as "x[] elements $\geq$ y" where x is a double sequence and y is a double scalar.

- SeqFloatGreaterThan: Represents double scalars with a property to each element of double sequences. Prints as "x[] elements > y" where x is a double sequence and y is a double scalar.

- SeqFloatLessEqual: Represents double scalars with a property to each element of double sequences. Prints as "x[] elements $\leq$ y" where x is a double sequence and y is a double scalar .

- SeqFloatLessThan: Represents double scalars with a property to each element of double sequences. Prints as "x[] elements < y" where x is a double sequence and y is a double scalar.

- SeqIndexComparison: Represents properties between elements of a long sequence and the indices of those elements. Prints as "x[i] [cmp] i" where [cmp] is one of $< \leq > \geq$.

- SeqIndexComparisonFloat: Represents properties between elements of a double sequence and the indices of those elements. Prints as "x[i] [cmp] i" where [cmp] is one of $< \leq > \geq$.

- SeqIndexNonEqual: Represents long sequences where the element stored at index i is not equal to i. Prints as "x[i] $\neq$ i."

- SeqIndexNonEqualFloat: Represents double sequences where the element stored at index i is not equal to i. Prints as "x[i] $\neq$ i."

- SeqIntComparison: Represents long scalars with a property to each element of long sequences. Prints as "x[] elements [cmp] y" where x is a long sequence, y is a long scalar, and [cmp] is one of the comparators $== < \leq > \geq$.

- SeqIntEqual: Represents long scalars with a property to each element of long sequences. Prints as "x[] elements $==$ y" where x is a long sequence and y is a long scalar.

- SeqIntGreaterEqual: Represents long scalars with a property to each element of long sequences. Prints as "x[] elements $\geq$ y" where x is a long sequence and y is a long scalar.

- SeqIntGreaterThan: Represents long scalars with a property to each element of long sequences. Prints as "x[] elements $>$ y" where x is a long sequence and y is a long scalar.

- SeqIntLessEqual: Represents long scalars with a property to each element of long sequences. Prints as "x[] elements $\leq$ y" where x is a long sequence and y is a long scalar.

- SeqIntLessThan: Represents long scalars with a property to each element of long sequences. Prints as "x[] elements $<$ y" where x is a long sequence and y is a long scalar.

- SeqSeqFloatEqual: Represents properties between two double sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] $==$ y[] lexically." If order doesn't matter for each variable, then the sequences are compared to see if they are set equivalent.

Prints as "x[] == y[]." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqSeqFloatGreaterEqual: Represents properties between two double sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] $\geq$ y[] lexically." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqSeqFloatGreaterThan: Represents properties between two double sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] > y[] lexically." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqSeqFloatLessEqual: Represents properties between two double sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] $\leq$ y[] lexically." If the extra information (e.g., order matters) doesn't match then no comparison is made at all. SeqSeqFloatLessThan Represents properties between two double sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] < y[] lexically." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqSeqIntEqual: Represents properties between two long sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] == y[] lexically." If order doesn't matter for each variable, then the sequences are compared to see if they are set equivalent. Prints as "x[] == y[]." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqSeqIntGreaterEqual: Represents properties between two long sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] $\geq$ y[] lexically." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqSeqIntGreaterThan: Represents properties between two long sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] $>$ y[] lexically." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqSeqIntLessEqual: Represents properties between two long sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] $\leq$ y[] lexically." If the extra information (e.g., order matters) doesn't match then no comparison is made at all. SeqSeqInt-LessThan Represents properties between two long sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] $<$ y[] lexically." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqSeqStringEqual: Represents properties between two String sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] == y[] lexically." If order doesn't matter for each variable, then the sequences are compared to see if they are set equivalent. Prints as "x[] == y[]." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqSeqStringGreaterEqual: Represents properties between two String sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] $\geq$ y[] lexically." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqSeqStringGreaterThan: Represents properties between two String sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] $>$ y[] lexically." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqSeqStringLessEqual: Represents properties between two String sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] $\leq$ y[] lexically." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- SeqSeqStringLessThan: Represents properties between two String sequences. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as "x[] < y[] lexically." If the extra information (e.g., order matters) doesn't match then no comparison is made at all.

- StringComparison: Represents lexical properties between two strings. Prints as "s1 [cmp] s2" where [cmp] is one of $== > \geq < \leq$.

- SubSequence: Represents two long sequences where one sequence is a subsequence of the other. Prints as "x[] is a subsequence of y[]."

- SubSequenceFloat: Represents two double sequences where one sequence is a subsequence of the other. Prints as "x[] is a subsequence of y[]."

- SubSet: Represents two long sequences where one of the sequences is a subset (each element appears in the other sequence) of the other. Prints as either "x[] is a subset of y[]" or as "x[] is a {sub,super}set of y[]" if x and y are set equivalent.

- SubSetFloat: Represents two double sequences where one of the sequences is a subset (each element appears in the other sequence) of the other. Prints as either "x[] is a subset of y[]" or as "x[] is a {sub,super}set of y[]" if x and y are set equivalent.

- UpperBound: Represents the property 'x $\leq$ c', where c is a constant and x is a long scalar.

- UpperBoundFloat: Represents the property 'x $\leq$ c', where c is a constant and x is a double scalar.

A.2 Program Point Slots

Each property appears at a certain point in the program, such as a procedure entry or exit. The following slots provide information about that program point:

- IsMainExit: States if the program point is the main exit of a method.
- NumValues: Reports the number of times this program point has been executed.
- PptSliceEquality: States if the program code contains equal variables equal.
- Arity: Reports the number of variables at this program point.
- IsPrestate: States if the program point is a precondition.
- PptConditional: States if this point is a conditional (only true for some inputs) program point.
- PairwiseImplications: States if this program point contains Implication properties.
- IsEntrance: States is this program point is a start of a method.

A.3 Program Variable Slots

Each property is dependent on some variables. For example, $x = y$ is dependent on variables x and y. The following slots provide information about the variables:

- NumVars: Reports the number of variables in this property.
- VarinfoIndex: Reports the index of the current variable. The slots for each variable are extracted in order of the variable index.
- IsStaticConstant: States if the variable is a constant static variable.
- CanBeMissing: States if a variable can be missing for some executions.

- PrestateDerived: States if a variable is a derived variable (e.g., a field of another variable) at the beginning of the method.
- DerivedDepth: States the depth of the derived variable (e.g., x.y has depth 1).
- IsPrestate: States if the variable value is at the beginning of a method. Note that the variable may be at the beginning of a method, while the property is not. For example, the property that states that variable x does not change within a method, is over two variables, x at the end of the method, and x at the beginning of the method.
- IsClosure: States if the variable is a closure.
- IsParameter: States if the variable is a parameter to the method.
- IsReference: States if the variable is passed to the method by value or reference.
- IsIndex: States if the variable is an index for a sequence.
- IsPointer: States if the variable is a pointer.

A.4 Property-Specific Slots and Other Slots

Slots listed below are either general slots about all properties, or specific to some properties. Some properties have slots that are only valid for those properties. For example, the LinearBinary property fits a line of type $y = ax + b$ between two variables x and y, so two of its slots are the values of a and b.

- CanBeEqual: States if variables of a property can be equal.
- CanBeLessThan: States if one variable of a property can be less than another variable.
- CanBeGreaterThan: States if one variable of a property can be greater than another variable.States if variables of a property can be equal.

- Probability: Reports the result of the null test hypothesis on the property.
- UnaryInvariant: States if the property is unary (over 1 variable).
- BinaryInvariant: States if the property is binary (over 2 variables).
- TernaryInvariant: States if the property is ternary (over 3 variables).
- Min: Reports the constant c for properties of type $x > c$ or $x \geq c$.
- Max: Reports the constant c for properties of type $x < c$ or $x \leq c$.
- NumElts: Sequence: Reports the number of elements in a sequence for all properties over sequences.
- Falsified: States if the property has been falsified. No falsified propertied should be reported to the user.
- HasCanonicalVariable: States if the property contains a canonical variable.
- HasNonCanonicalVariable: States if the property contains a non-canonical variable.
- Iff: States if the property is bidirectional (contains an iff).
- EnoughSamples: States if the property has seen enough samples to be statistically justified.
- IsExact: States if the property is exact.
- IsObvious: States if the property is obvious.
- IsObviousStatically: States if the property is obvious because of static analysis.
- IsObviousDynamically: States if the property is obvious because of dynamic analysis.
- IsWorthPrinting: States if the property is worth printing, as defined by Daikon filters.

- IsImplied: States if the property is implied by another property.
- IsInteresting: States if the property is interesting, as defined by Daikon filters.
- HasUninterestingConstant: States if the property contains an uninteresting constant.
- IsJustified: States if the property is statistically justified.

Bibliography

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

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977.

[3] The Daikon Invariant Detector User Manual. `http://pag.lcs.mit.edu/daikon/download/doc/daikon.html`, August 2003.

[4] William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE 2001, Proceedings of the 23rd International Conference on Software Engineering*, pages 339–348, Montreal, Canada, May 16–18, 2001.

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

[6] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the*

22nd International Conference on Software Engineering, pages 449–458, Limerick, Ireland, June 7–9, 2000.

[7] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, April 2001.

[8] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *10th International SPIN Workshop on Model Checking of Software*, pages 121–135, Portland, Oregon, May 9–10, 2003.

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

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

[11] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, Sorrento, Italy, May 16–21, 1994.

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

[13] Ross Quinlan. Information on See5 and C5.0. http://www.rulequest.com/see5-info.html, 2003.

[14] Ryan Michael Rifkin. *Everything Old Is New Again: A Fresh Look at Historical Approaches in Machine Learning.* PhD thesis, MIT Sloan School of Management Science, Cambridge, MA, September 2002.

[15] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.

[16] David Saff and Michael D. Ernst. Can continuous testing speed software development? In *Fourteenth International Symposium on Software Reliability Engineering*, Denver, CO, November 17–20, 2003.

[17] Gerard Salton. *Automatic Information Organization and Retrieval.* McGraw-Hill, 1968.

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

[19] Filippos I. Vokolos and Phyllis G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the International Conference on Software Maintenance*, pages 44–53, 1998.

[20] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley, third edition, 1992.

[21] Yichen Xie and Dawson Engler. Using redundancies to find errors. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 51–60, Charleston, SC, November 20–22, 2002.