

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Predicting Fault Locations from Failures Using a Machine Learning Classifier

A thesis submitted in partial satisfaction of the requirements for the degree
Master of Science

in

Computer Science

by

Geoffrey Compton Hulette

Committee in charge:

Sorin Lerner, Chair
Sanjoy Dasgupta
Ranjit Jhala

2007

Copyright
Geoffrey Compton Hulette, 2007
All rights reserved.

The thesis of Geoffrey Compton Hulette is approved:

Chair

University of California, San Diego

2007

TABLE OF CONTENTS

Signature Page	iii
Table of Contents	iv
List of Figures	vi
List of Tables	vii
Acknowledgments	viii
Abstract	ix
Chapter 1. Introduction	1
1.1 Mutation Analysis	2
1.2 Machine Learning	3
1.3 Method Summary	3
1.4 Definitions	4
1.5 Outline	5
Chapter 2. System Overview	7
2.1 Failure Artifacts	7
2.2 Source Code	8
2.2.1 Realistic Faults	9
2.2.2 UNL Software Artifact Infrastructure Repository	9
2.3 Mutation Analysis	10
2.3.1 Mutant Operators	11
2.3.2 Polyglot	11
2.3.3 FWGrid	11
2.4 Machine Learning	13
Chapter 3. System Implementation	14
3.1 The Mutator	14
3.1.1 Implementation	14
3.1.2 Adding Mutation Operators	16
3.1.3 Invoking Mute	16
3.2 Source Code Preparation	17
3.2.1 Ant Targets	17
3.2.2 Modifying JUnit	18
3.2.3 Injecting Realistic Faults	18
3.2.4 Preprocessing with Polyglot	19
3.2.5 Testing a Mutation	19

3.3	Using FWGrid for Mutation Analysis	20
3.3.1	Iterating Over All Mutations	20
3.3.2	Using the Cluster	21
3.4	Generating Test Data	22
3.5	Processing Results from the Analysis	22
3.5.1	Building an ARFF Database	23
3.6	Applying WEKA to Processed Data	24
3.6.1	Naïve Bayes Top- k	25
Chapter 4. Starting with a Correct Program		27
4.1	Experimental Setup	27
4.2	Results	28
4.3	Evaluation	29
Chapter 5. Starting with a Faulty Program		32
5.1	Experimental Setup	32
5.2	Results	33
5.3	Evaluation	34
Chapter 6. Discussion		35
6.1	Mutation Analysis	35
6.2	Machine Learning	36
6.3	Source Code Locations	37
6.4	Synthetic vs Real Faults	38
Chapter 7. Related Work		40
7.1	Locating Faults	40
7.1.1	Slicing	40
7.1.2	Change Classification	41
7.1.3	Delta Debugging	41
7.1.4	Program Invariants	42
7.1.5	Machine Learning on Program Executions	43
7.2	Bugs for Testing	44
Chapter 8. Conclusion		45
8.1	Summary of Results	45
8.2	Future Work	46
8.2.1	Improving Accuracy	46
8.2.2	Other Work	46
References		47

LIST OF FIGURES

Figure 4.1: Results on JMeter using Naïve Bayes Top- k , $\forall k \in \{1, 3, 5\}$, and a trivial predictor	29
Figure 4.2: Summary of results for $k = 5$, broken down by exception	31

LIST OF TABLES

Table 2.1: Implemented mutation operators	12
Table 3.1: Example of project-unique mutation IDs	21
Table 4.1: Source code statistics for JMeter	28
Table 4.2: Exception classes observed in JMeter failures	30
Table 5.1: Number of realistic bugs where the analysis revealed a failure matching an original failure, but in a different test case	34

ACKNOWLEDGMENTS

I would like to gratefully acknowledge the extensive help and encouragement of Professor Sorin Lerner and Professor Brian Demsky. I would also like to acknowledge the help of my committee members, Professor Sanjoy Dasgupta and Professor Ranjit Jhala, as well as my many colleagues at UCSD.

This research was supported in part by the UCSD FWGrid Project, NSF Research Infrastructure Grant Number EIA-0303622.

ABSTRACT OF THE THESIS

Predicting Fault Locations from Failures Using a Machine Learning Classifier

by

Geoffrey Compton Hulette

Master of Science in Computer Science

University of California, San Diego, 2007

Sorin Lerner, Chair

The first step in finding a bug is knowing the failures it causes. Failure information is usually the first and often the best information a programmer gets.

This thesis describes a method for predicting the location of a fault, given stack traces of failures it causes. We use Naïve Bayes, a machine learning algorithm, trained on the details of failures and locations of related faults, to predict the fault locations. To obtain sufficient quantities of faults and failures, we use a mutation analysis, optimized for parallel execution on the FWGrid cluster at UCSD. The Naïve Bayes algorithm uses data from this analysis to generate a statistical model of the failures we observe, and this model is then used to predict the fault locations of newly observed failures.

In one experiment, we use an implementation of this method to analyze JMeter, a Java project containing almost 50,000 lines of code in around 250 files. The implementation correctly locates 40% of the faults we tested to within the correct source file, and 20% to within the correct code block.

In a second experiment, we investigate a more realistic variation of our method

that can be trained from the mutation analysis of a program that already contains a real fault. This variation is shown to be potentially viable, and deserving of further research.

Chapter 1.

Introduction

The first step towards locating a bug is knowing how it causes a program to fail. The details of the failure – what happened? at what point in the execution? was an error code produced? – are usually the first and best information a programmer gets in the debugging process. There is often an intuitive link between a program failure and the faulty code that caused it, at least for programmers who are intimately familiar with the code.

In this thesis, we describe a method for automatically inferring the location of a fault from details of failures caused by that fault. In experiments with Apache JMeter, a Java project containing almost 50,000 lines of code in around 250 source files, this method was able to correctly locate 40% of the faults we tested to within the correct source file, and 20% to within the correct code block.

The method uses a machine learning algorithm to classify the details of failures by the location of the related fault. To obtain a sufficient quantity of faults and failures with which to train the algorithm, we run a mutation analysis on the program's source code. The machine learning classifier uses this analysis to generate a statistical model of the failures we observe, and this model is used to predict the fault location of newly observed failures.

The first implementation of our method relies on having the correct source code

(i.e. code that does not cause any failures in its tests) for the program being analyzed. We recognize that this requirement is unrealistic – if a developer already has a correct program, they do not need to locate faults. So, a second implementation starts with a program containing the fault we want to find. We present preliminary results indicating that this implementation may be viable.

1.1 Mutation Analysis

Mutation analysis is a process by which a program is systematically altered with the intent of introducing faults [1][32][6][24]. The altered program is called a mutant, and the method by which a particular kind of alteration is introduced is called a mutation operator. A single mutation operator may be applied at different locations in the source code. For example, a mutation operator may be defined that changes an integer increment operation to a decrement, and may be applied anywhere an increment operator appears in the program. A mutant is the result of applying a single mutation operator to a single location in the original source code. For a given set of mutant operators M , and a program P , there is a set of possible mutants $M(P)$ that can be created by applying each mutant operator in M to every applicable location in P . A full mutation analysis generates each mutant in $M(P)$ and compiles, tests, and records any resulting failures for each one.

Mutation analysis was invented to reveal inadequacies in test cases. If a mutant passes all its tests (i.e. no failures are observed), then either the mutant is functionally equivalent to the original program, or the tests are inadequate and need to be extended or modified [1]. In this project, we instead, use mutation analysis to generate sets of faults and failures with which to train our machine learning classifier.

1.2 Machine Learning

The field of machine learning is concerned with algorithms that can extrapolate from previously observed data to make predictions about new data. We are particularly interested in a machine learning problem known as classification [13][7]. Classification algorithms (“classifiers,” hereafter) “learn” by observing a set of example data points, where each data point has some set of features, and also a label. After training, the classifier attempts to predict labels for unlabeled data points. For example, a data set might be drawn from a group of people, with the features being the heights and weights of each person, and the label being “male” or “female.” After training, the classifier could examine the height and weight of a new person, and make a guess as to whether the person is a man or a woman. A classifier’s guesses (sometimes called classification predictions, or “predictions”) have a certain probability of being correct. This probability depends on the particular features of the data point in question, the quality and quantity of the training data, and the classifier algorithm. A second set of labeled data points (i.e. data points not in training) can be used to evaluate the accuracy of the trained classifier by comparing its predictions against the known labels.

Classifiers usually work by generating a probabilistic model of the training data [13][7]. A classifier’s model should be as accurate as possible with regard to the data points being predicted upon. Note that this goal is not necessarily the same as maximizing accuracy with regard to the training data.

In this project, our data points will be program failures. The features of a data point will be drawn from information about the failure, and the label will be the location of the fault that caused the failure. In this way, our method allows us to predict the location of a fault from information about the failure.

1.3 Method Summary

All of our experiments will involve variations on the following steps.

1. We select a program to examine. The program must provide a fairly comprehensive test suite for this method to be effective.
2. A mutation analysis is performed. This generates many mutant copies of the code, each containing a single mutation. We record the location of each mutation.
3. Each mutant is compiled and run through its test suite. We record the details of any observed failures.
4. We train the classifier. To construct each data point in the training set, we use the mutation's location as the label, and details extracted from the failure as its features. The result of this step is a statistical model that hopefully represents relationships between failure features and fault locations.
5. We evaluate the trained model. This is done by first repeating the procedure of inserting faults and generating failures, but using a set of realistic faults in place of mutations.
6. We predict the locations these realistic faults using the classifier, and compare the predictions against their known locations. The ratio of correct predictions to the number of faults tested represents the accuracy of the classifier, and is the benchmark we use to evaluate the method.

1.4 Definitions

Fault An area in the source code of a program that does not perform as the programmer intended. A single fault is located in a source code file, and may span multiple lines of code.

Bug A synonym for “fault.”

Error An incorrect program state, i.e. the data structures or control flow of the program were intended to be a certain way, but are not. Errors are usually caused by faults, but may also be caused by other problems – broken hardware, for example.

Failure A manifestation of an error that is visible to the user or programmer. Error messages or crash logs are examples of failures.

Failure artifact Here, a failure artifact is any data associated with a failure that is not the program source code. This may be input files, core dumps, output, and so on. In this project we are especially interested in the stack traces printed by Java exceptions as failure artifacts.

Mutation A single, synthetic alteration injected into a program’s source code, intended to cause an error that may reveal a failure. A mutation is located in a source code file, and may span multiple lines of code. Note that in this thesis the term “fault” is sometimes used interchangeably with “mutation”, since for this project mutations are intended to model faults.

Mutation Operator An operator that implements a transformation of the original source code. The kind of operator determines the details of the transformation. For example, the NumericBinary mutation operator is a kind of mutation operator (see Table 2.1).

Mutator A program that injects mutations using a set of mutation operators.

1.5 Outline

The remaining chapters of this paper are organized as follows. Chapter 2, *System Overview* provides more details on our method. Chapter 3, *System Implementation* describes an implementation of this method. Chapter 4, *Starting with a Correct Program* discusses the results of testing our implementation on a correct program (i.e. one that does not cause any failures). Chapter 5, *Starting with a Faulty Program* details

our preliminary experiments using a variation of our method where the program used in training already contains a fault. In Chapter 6, *Discussion*, we reflect on lessons learned over the course of this project. We also look at some alternatives to choices made in the implementation. Chapter 7, *Related Work* presents a review of other research having similar goals. Finally, Chapter 8, *Conclusion* summarizes of the results of our work, and discusses ideas for future work.

Chapter 2.

System Overview

2.1 Failure Artifacts

There are many different kinds of failure artifacts produced by different programs. We decided to use the stack traces printed by Java program by default whenever an uncaught exception is thrown. Stack traces have a number of benefits. They are generated by many different programs – almost all Java programs, in fact. Stack traces are easy to capture by redirecting the program’s error stream to a file. Stack traces have a regular structure that can be parsed. Finally, stack traces are easily separated into different parts (i.e. files, methods, line numbers) which can be used as the features of a data point for training the classifier. See Section 2.4 for more details on how the parts are separated and used.

Most importantly, we guessed that a stack trace might be a useful source of information for the classifier, since we expect some elements of the stack trace to be correlated with the fault location. Consider an example where a method X returns an integer value. Method X is called by a method Y , which uses the integer value as an index into an array. If we mutate method X , it may return an incorrect index, causing method Y to fail when it tries to access an out-of-bounds array index. The resulting stack trace, on its own, does not contain a reference to method X , but if many mutations

in X cause similar failures in Y , the classifier learns the correlation. This example is made simple to illustrate the principle – in practice the classifier can model much more complex relationships.

2.2 Source Code

An important goal of this project is testing our method in a realistic context. So, in our experiments we analyze programs large and complex enough to make locating bugs in them by hand a tedious process. We judge this to be programs having at least thousands or preferably tens of thousands of lines of code.

Our method only applies to deterministic, single-threaded programs. Concurrent programs or code containing randomized algorithms can exhibit different failure behavior in multiple runs of the same test case. This would create, in effect, noise in the training data, making our trained model less reliable.

Our analysis currently only works on Java programs. One advantage of Java is its static type system, which was helpful for implementing the mutator (see Section 3.1.1). There is no reason our method should not be able to work with other programming languages as well.

For our mutation analysis, a program needs a comprehensive set of unit tests. We use these as a way of revealing failures. Java has a very popular unit test framework called JUnit. Almost all modern Java programs include a set of tests for JUnit, and so it became our de facto official testing framework.

The source code for our experiments is taken from packages provided by the UNL Software-artifact Infrastructure Repository (SIR). See Section 2.2.2 for more details on the SIR.

2.2.1 Realistic Faults

To measure the accuracy our trained machine learning model, we test its predictions against known locations of realistic faults. Ideally, we would have liked to use bugs that had actually been found (and fixed) by the developers as they were developing the program. In fact, examples of such bugs are surprisingly hard to find. Instead, we settled for testing our model on the realistic bugs that were provided as part of the source code packages from the SIR. See Section 2.2.2 for more information on the SIR.

Any faults that we use for testing need to be recorded in a way that facilitates easy injection into the source code. The bugs in SIR satisfy this requirement. See Section 3.2.3 for details on how this is done in our implementation.

Instead of real bugs, or the limited set of bugs in SIR, we could have used synthetically generated faults for testing. Ultimately, we rejected this idea. Initially, testing with synthetic faults seemed an appealing option – we generate synthetic faults anyway for our mutations, and we could easily reserve some of those for testing. We decided, however, that testing on our own mutations would be biased. One goal of this project is to evaluate whether our method could be useful to real programmers, and this implies that we should test on the most realistic bugs we can find.

We considered other sources of real bugs before settling on the SIR. These are discussed in Chapter 7.2.

2.2.2 UNL Software Artifact Infrastructure Repository

As described in Section 2.2 and Section 2.2.1, we test our method on Java programs that include source code, unit tests, and a set of realistic faults. The UNL Software-artifact Infrastructure Repository (SIR) conveniently provides packages including all these items [12]. The SIR contains, among other things, several versions of source code for open-source Java programs, along with build scripts, dependencies, unit tests, and realistic bugs for each version. Unfortunately, the bugs were not created during the program’s development. Instead, they were created by researchers at

UNL, who designed them to resemble bugs made during development. Using the SIR database saved us a great deal of time and effort that would have been spent collecting and organizing programs ourselves.

2.3 Mutation Analysis

A mutation operator is the basic unit of mutation analysis. Each mutation operator defines rules that describe necessary features of code where the operator can legally be applied. When a legal region of code is found, and the mutation operator applied, the result is a well-defined transformation of the code at that point. Conceptually, mutation is equivalent to injection of a synthetic fault. Examples of mutation operators include “change addition to subtraction in an integer expression,” or “replace `return var`, where `var` is some string expression, to `return null`.”

Ideally, the set of mutation operators would be identical to the most common set of faults that programmers accidentally introduce into their code. In this ideal situation, our classifier model would be trained on information closely resembling real faults and failures, and might therefore perform well at predicting real faults and failures. In practice, accurately modeling the wide variety of faults that appear in real programs is difficult [1], and the set of mutation operators is an approximation of this ideal. The mutation operators we implement are listed in Section 2.3.1.

The process of injecting mutations, compiling, and testing thousands of mutant programs is computationally expensive. Therefore, the total number of mutations should be as many as the available hardware can manage in a reasonable timeframe, but no more. We found that, using the set of mutant operators described in Section 2.3.1 and running the analysis in parallel on FWGrid (see Section 2.3.3), our system was able to process every possible mutant of a program having about 50,000 lines of code in approximately ten hours. That timeframe is significantly longer than would be tolerated during interactive development, but for this initial investigation speed was a low priority.

Currently, our method only generates and tests first-order mutants (i.e. mutants

that contain only one alteration of the original program)[1].

2.3.1 Mutant Operators

A common difficulty with mutation analysis is deciding which mutation operators to use [1][32]. Table 2.1 shows the mutation operators we implement. We use this particular set for a number of reasons. First, as discussed in Section 2.3, we want mutations that resemble real faults as closely as possible. The mutation operators we implement do not model all possible faults, but we believe that each one models a mistake that programmers can and do make. Finally, having regions of code where no mutation operators can be applied is undesirable – in order to predict a fault in a particular location, we need to have observed the effects of mutating that location. So, we chose a set of mutation operators such that at least one can be applied to almost every line of code in the programs we test. Finally, there are well-established sets of mutation operators [1][32], and our set includes many of these.

2.3.2 Polyglot

We use Polyglot [31], an extensible Java compiler, as a framework for our mutator implementation. Being able to leverage the power of a full compiler is a huge benefit – without it the mutator would not have complete access to Java’s types, which are difficult to infer from a more basic code analysis. This enabled us to write a mutator that could perform sophisticated mutations. See Section 3.1.1 for details on the mutator implementation.

2.3.3 FWGrid

Mutation analysis is expensive in terms of time. The time to solution is guaranteed to increase linearly with the number of mutations analyzed, and each mutation must be injected it, compiled, tested, and the failures recorded and processed. However,

Table 2.1: Implemented mutation operators

Mutation Name	Description
NumericBinaryOperator	Applied to numeric (<code>float</code> , <code>double</code> and <code>int</code>) binary expressions. Replaces an addition (+), subtraction (-), multiplication (*) and division (/) operators with a different operator from that same list.
LogicalBinaryOperator	Applied to logical (<code>boolean</code>) binary expressions. Replace “and” (&&) or “or” () operators with a different operator from that same list.
ComparisonBinaryOperator	Applied to numeric (<code>float</code> , <code>double</code> and <code>int</code>) binary expressions. Replaces “equals” (==), “not equal” (!=), “greater than” (>), “less than” (<), “greater than or equal to” (>=), “less than or equal to” (<=) operators with a different operator from that same list.
ExprToZero	Replaces a numeric expression (or sub-expression) with zero.
AssignNull	Replaces the right-hand side of an object assignment expression with <code>null</code> .
PassNull	Replace an object expression that is a parameter to a method invocation with <code>null</code> .
ReturnNull	Replaces a <code>return</code> expression that returns an object with <code>return null</code> .
UnaryBoolean	Replaces a logical expression (or sub-expression) with its inverse.
VariableName	Replaces a use of or assignment to some variable with a different variable of the same type that is also in scope.
MethodCall	Replaces the name of a method in a method call to another method in the same object (or its superclass) having the same arguments.

because each mutation can be executed independently of the others, we were able to take advantage of a cluster computer at UCSD called FWGrid [21]. FWGrid has 128 processor nodes connected on a gigabit network and a large shared storage space for data. Process scheduling is managed with Sun Grid Engine software [36]. Running a full analysis on a single program for the experiments described in Chapter 4 on FWGrid takes around 8 hours. Without the parallelism afforded by FWGrid, the same analysis would have taken almost 42 days. The FWGrid cluster made these experiments practical.

2.4 Machine Learning

There are many different machine learning algorithms for classification. We implement a variation on an algorithm called Naïve Bayes [13]. Naïve Bayes is conceptually simple, and easy to implement and apply. It has also been shown to perform well on classification problems [20]. In many cases its performance can be superior to more sophisticated techniques.

Naïve Bayes uses a probabilistic model to represent the training data. One benefit of this representation is that Naïve Bayes can give us, along with the predicted label, a ranking reflecting the model's confidence in that prediction. We exploit this property in our implementation, as explained in Section 3.6.1.

Machine learning works best when there is sufficient training data available for the algorithm to construct an accurate model [13]. From our mutation analysis, we have a large quantity of data, but the quality of the data (i.e. the resemblance of our mutations to real bugs) may be problematic.

Chapter 3.

System Implementation

3.1 The Mutator

We call our mutator “Mute.” Mute is implemented in Java as an extension of Polyglot version 1.3.4, available from [34].

Polyglot provides functionality to read and parse Java files, converting them to an abstract syntax tree (AST) with type information. The default behavior is to print the AST as Java source code, effectively implementing an isomorphic source-to-source Java translation.

Polyglot can be extended by inserting phases into the compilation. For example, to add syntactic sugar to Java, a programmer might start by inserting a lexical analysis phase that can read the new syntax. Further details on extending Polyglot can be found in [33].

3.1.1 Implementation

To implement Mute, we insert a compilation phase after the AST is generated, but before Java source code is printed. Mute assigns an identification number to every possible location in a source file where a mutation can be applied. This number is called a “Mutation ID.” Mutation IDs are unique per Java file and start at zero. If Mute finds

that a Java file has n possible mutations, it will assign them the ID numbers $0, 1 \dots n-1$. A greater ID number does not necessarily correspond to a later location in the source file. It is guaranteed that Mute will assign a given mutation the same ID number each time it is invoked on the same source file.

Mute expects a Java source file, a mutation ID, and an output directory as arguments. It injects the mutation referenced by the ID into the source file. The resulting mutant is written to the output directory, using the original Java package directory structure and file name. Mute can also be invoked without a mutation ID argument. In this case no file will be written, and Mute will print a list containing IDs and locations for all possible mutations it finds in the source file.

A mutation is represented by a mutation operator and a location in the source file where that operator is applied. Mute encapsulates mutation operators with a class called `AbstractMutation`. Subclasses of `AbstractMutation` override the abstract `mutate` method to implement the mutation operator. For example, we implement an `OperatorReplacementMutation`, `AssignNullMutation`, and so on. Section 2.3.1 has a complete list of our mutation operators. Mute internally represents a mutation location using a node in the AST. AST nodes have the type `polyglot.ast.Node` in `Polyglot`, and encapsulate the location information Mute needs. When an `AbstractMutation` subclass is instantiated, a node is passed to the constructor to indicate the location where the mutation is to be applied. Invoking `mutate` modifies the node in place according to the rules of the mutation operator.

Mute uses a class called `Mutator` to build the list of possible mutations. This phase begins after `Polyglot` has finished building the AST. The `Mutator` object uses `Polyglot`'s visitor pattern [22] utility class to iterate over each node in the AST. This process enumerates mutations using the following algorithm.

1. Initialize an empty list to store the mutations.
2. Iterate over each node in the AST using `Polyglot`'s visitor utility.
3. Check each node's type, and the number and types of its children, to see if they

match any patterns we can mutate. For example, if the node is an arithmetic operator node, and its type is `int`, it can be mutated.

4. For each match in step 3, instantiate the appropriate subclass of `AbstractMutation` to represent the mutation, and append it to the end of the list. Note that it is possible for more than one mutation to be applicable at a single node. For example, an integer addition may be mutated to subtraction, multiplication, or division.

The result of this process is a complete list of possible mutations for the input file. A mutation's ID number is its position in the list.

3.1.2 Adding Mutation Operators

There are two steps in creating a new mutation operator for Mute. First, subclass `AbstractMutation`. The subclass must override `mutate`, which transforms the AST node. The node is accessible to `mutate` through an instance variable, which is assigned in the object's constructor. Second, modify the `Mutator` class to locate legal AST nodes for the mutation operator. When a legal node is found, instantiate the subclass and add it to the list.

3.1.3 Invoking Mute

Mute is invoked using a shell script called `mute-local`. The script takes two optional arguments: `-d dir` where `dir` is the directory where Mute writes mutated Java files, and `-muteid id`, where `id` is the mutation ID. The `-muteid` argument may be omitted, causing Mute to print out a list of all possible mutations. Mute also accepts most standard Java compiler switches, such as `-cp` to specify a classpath. The last argument is a Java source file.

3.2 Source Code Preparation

In our experiments, we analyze source code packages from the SIR (see Section 2.2). We could, however, modify our implementation to analyze any Java program that meets the criteria in Chapter 2. This section explains how we use the programs from the SIR.

3.2.1 Ant Targets

Ant [2] is a tool that, like Make [29], automates compilation and build processes. Unlike Make, it is targeted towards Java programs. Ant is invoked using the command `ant` followed by a “target” that describes the task Ant is to perform. A target may have dependencies that cause other targets to be invoked first.

All the projects in the SIR that we use include Ant scripts that compile and test them. In some cases we modified these scripts slightly to conform to our computing environment.

Ant must be installed to use our implementation. To take advantage of FWGrid, Ant must also be available from each cluster node. Detailed instructions for installing and configuring Ant are available online [3].

Our implementation expects the command `ant all` to build a project. The build should include compilation of the main program and dependencies, and produce any binary files needed to run the unit tests.

Our implementation also expects an `ant clean` command to clean up the build environment, removing files generated during the build process and restoring the project to its original state.

Finally, `ant test` must execute all the program’s unit tests, and write the summary output to a file named `testsuite.out` in the root project directory.

3.2.2 Modifying JUnit

JUnit is our implementation’s unit test framework (see Section 2.2). In our experiments we use a slightly modified version of JUnit. The original version halts a test’s execution immediately when an assertion fails, which may prevent failures after the assertion from being revealed. We modified JUnit to allow a unit test to continue execution past assertion failures.

3.2.3 Injecting Realistic Faults

Each program package in the SIR features several injectable bugs created by UNL researchers and intended to simulate bugs that could have been written during development.

These bugs are encoded using C preprocessor directives. Some Java files in the project are duplicated in a C file having the same name (except having the suffix “.c” instead of “.java”), and residing in the same directory as the original file. These files contain the same Java source code, except for the addition of a section containing the code for the bug. This section is surrounded with an `#ifdef` compiler directive. The bug is activated by invoking the C preprocessor with a command line option that enables a `#define` specific to one of the bugs. The `#define` values for each bug are listed in a file called `faults.h` in the root project directory.

In our implementation, we pre-generate the Java source code for each bug. We store the generated faulty files for each bug in a directory `src_faults/fault_XXX`, where “XXX” is a number identifying the bug. The faults are enumerated in the same order as their corresponding `#define` codes in `faults.h`. The implementation includes a script called `activate-fault.rb` that, when invoked with a bug number, will activate the bug by copying the appropriate faulty files over the original ones. Calling `activate-fault.rb` with no fault number will restore the original files, removing the bug.

3.2.4 Preprocessing with Polyglot

As mentioned in Section 6.3, our implementation must take care that lines of code in mutated source files retain the line numbers they had in the original source files. Otherwise, the classifier will be trained and tested using locations that do not correspond, causing incorrect predictions. This problem arises because Polyglot does not preserve the original line number structure of files it processes – comments are removed, and whitespace is reproduced in a non-standard way. The result is that Mute prints code using line numbers that do not match the original, hand-coded files.

We solve this problem by pre-processing each Java file with Polyglot’s default source-to-source translation. This ensures that the files Mute reads have the same line number structure as the mutants it writes. Polyglot includes a script called `jlcc` that executes the source-to-source translation.

3.2.5 Testing a Mutation

Once we have prepared the source code, we run the mutation analysis. The analysis injects mutations and then compiles and tests the mutants. This process executes the following steps.

1. Select a Java file and mutation ID.
2. Copy the source file from its original location to a backup directory.
3. Invoke `mute` on the original file with the selected mutation ID. Copy the mutant file into the project source directory, overwriting the original file.
4. Invoke `ant all` to build the mutant program.
5. Invoke `ant test` to execute the unit tests and write the results to `test-suite.out`.
6. Copy `testsuite.out` to a results directory, along with any information about the current mutation.

7. Invoke `ant clean` to remove the build files.
8. Copy the original source file from the backup directory back into its original location, overwriting the mutant file. This restores the project to its original state, and the process can be repeated.

3.3 Using FWGrid for Mutation Analysis

Using the set of mutation operators from Section 2.3.1, the total number of possible mutations in a project is on the order of the number of lines of code in the project. The programs we examine in our experiments have thousands or tens of thousands of possible mutations. Since testing each mutation may take several minutes on a modern single-processor machine, a complete analysis can be very time consuming. We address this problem by adapting the analysis to take advantage of the FWGrid cluster (see Section 2.3.3). Note that mutation analysis is inherently parallel – each mutation can be injected, compiled, and tested independently from the others.

Using FWGrid is straightforward. Users are allowed to log into a “head node,” to access shared storage, run programs, and submit jobs to a scheduler. The cluster nodes are not directly accessible from the head node – the user may only submit jobs, after which the scheduler manages when and on which nodes the jobs execute. After submitting a job, the operator either waits for the job to complete or requests that it be terminated.

3.3.1 Iterating Over All Mutations

To run a mutation analysis on FWGrid, we start a master process (called `Gridrun`) that executes on the head node. `Gridrun` enumerates all the mutations in the project, and then submits each mutation as a job to the SGE scheduler. Each mutation job executes one complete mutation, as described in Section 3.2.5, and saves the test results into the shared storage space.

Table 3.1: Example of project-unique mutation IDs

Global ID	File	Mutation ID
0	A.java	0
1	A.java	1
2	A.java	2
3	B.java	0
4	B.java	1
5	C.java	0
6	C.java	1
7	C.java	2
8	C.java	3

For Gridrun to enumerate each mutation in the project, it first finds every Java source file in the project. Then, Gridrun invokes Mute on each file, omitting the mutation ID argument, to get the number of mutations. The file name and the number of possible mutations in it are appended to a new line in a file named `mutations`. Gridrun then iterates through each line of `mutations` and assigns each file and mutation ID a new, project-unique ID, based on the following algorithm.

1. Initialize a list to hold the project-unique IDs.
2. Load `mutations` and iterate through the list. For every possible local mutation in a file, append a new global ID number to the list project-unique IDs, along with the file name and the mutation ID it corresponds to.

As an example, suppose there are files `A.java`, `B.java`, and `C.java`, having 3, 2, and 4 mutations respectively. Gridrun would assign them the project-unique mutation IDs shown in Table 3.1.

3.3.2 Using the Cluster

Once it has assigned a project-unique ID to each mutation, Gridrun iterates through them, submitting mutation jobs to the SGE scheduler. A mutation job copies

the entire project to local storage on its assigned compute node and injects, builds, and tests the mutation. Finally, the job copies the test results, along with logs of the output from Mute and the compiler, to an output directory in the cluster’s shared storage space. The job then terminates, and the scheduler assigns a new mutation job to the node.

3.4 Generating Test Data

In addition to mutations, we need to inject and test the realistic bugs from the SIR. The results will be used in test data points for evaluating the trained classifier. We wrote a script called “Runtests” that iterates through each of the realistic bugs, injects each one (see Section 3.2.3), and runs the unit tests. The results are saved in a results directory. We did not adapt Runtests for the cluster, because the SIR projects have comparatively small numbers of realistic bugs (Table 4.1 shows the number of bugs for each project).

Runtests also records, for each realistic bug, the location (i.e. the file name and line number) where the bug was inserted. This information is obtained by parsing the output from a standard UNIX `diff` of the source tree containing the inserted bug against the original source tree. The locations of the realistic faults are needed to verify the predictions of the classifier.

3.5 Processing Results from the Analysis

The result of a complete mutation analysis is a set of directories, one for each mutation, and also one for each realistic bug. Each directory contains results from the unit tests.¹Each directory also has a file describing the location of injected fault – this file contains `diff` output for realistic bugs (see Section 3.4), and the Mute log output for mutations.

¹If an error prevents the completion of a mutation, the directory for that test will be empty or may contain only error logs. These directories are ignored. For example, sometimes mutant programs do not terminate. In this case the mutation job would pass a time limit, be terminated, and an error log would be

We parse data from the files in these directories to build a data set used by the classifier. See Section 3.5.1 for details on what data is parsed.

3.5.1 Building an ARFF Database

Our WEKA-based machine learning classifier expects training and test data sets in the ARFF database format. An ARFF database is a text file where rows of text represent observations, and each row is divided into comma-delimited columns encoding the label and features of each observation. There is also a header section that describes the format of the columns.

In our implementation, each row in the ARFF database will correspond to one observed failure (i.e. a single stack trace) from either a mutant or a realistic bug. As shown below, realistic bugs are encoded slightly differently from mutants, so that they can be distinguished within WEKA.

The script that builds the ARFF database accepts an argument specifying the maximum number of stack trace entries to include. This number is n in the descriptions below. If the script encounters a stack trace with fewer than n lines, the missing data is represented in the database with the string “NA”.

Each column in our ARFF database is described below.

Observation Kind This column is assigned a value of either “TRAIN” or “TEST”, indicating the source of the observation was a mutation or realistic bug, respectively.

Mutation ID or Realistic Bug Code This column contains either a project-unique mutation ID indicating which mutation caused the failure or, in the case of a realistic bug, the `#define` code that activates the bug.

Fault Location For a mutation, the value of this column is the location of the mutation encoded as a string with the Java file name and two line numbers. The line

written to the results directory.

numbers indicate the first and last lines of the code block surrounding the mutated code (see Section 6.3). For a realistic bug, the value is always the same placeholder string: “????”.

Exception class The value of this column is a string containing the name of the exception’s Java class (for example `java.lang.NullPointerException`).

First trace file This column contains the file name of the first entry on the stack trace.

First trace line number This column contains the line number of the first entry on the stack trace.

Second trace file This column contains the file name of the second entry in the stack trace.

Second trace line number This column contains the line number of the second entry in the stack trace.

...

n^{th} **trace file** This column contains the n^{th} file name on the stack trace, where n is the maximum specified size for the stack trace.

n^{th} **trace line number** This column contains the n^{th} line number on the stack trace.

3.6 Applying WEKA to Processed Data

Using a WEKA machine learning classifier algorithm involves the following steps:

1. Load the ARFF-formatted database (see Section 3.5.1).
2. Divide the data into two sets: one for training and one for testing.
3. Designate a column as the label of the data set (see Section 2.4).

4. Create a model of the observed data by presenting the classifier learning algorithm with observations from the training data set. The details depend on the algorithm but, generally, if the training data is statistically representative of the underlying distribution, then the more examples that are used for training, the more useful the predictions of the generated model will be [13].
5. Use the trained model to predict the labels of data in the test set.
6. Compare the predictions against the correct labels. The quality of the machine learning model is usually expressed as the ratio

$$\frac{\text{accurate predictions}}{\text{total predictions}}$$

We wrote a Java program called “TrainTest” that uses WEKA and executes these steps. TrainTest uses failures from the mutation analysis (data where the Observation Kind is TRAIN in the ARFF file) as the training data set, and failures caused by the realistic bugs (where Observation Kind is TEST) as the test data set. The label is the fault location.

Once TrainTest loads the observations and separates them into training and test sets, we discard the Observation Kind and the Mutation ID/Realistic Bug Code columns, as they are not relevant to the training.

3.6.1 Naïve Bayes Top- k

WEKA provides several classifier algorithms. We focus on one in particular, called Naïve Bayes [20][15].

Usually, Naïve Bayes is used to predict only the single most likely classification. In our implementation, we use WEKA’s Naïve Bayes algorithm to report the top k most likely locations in decreasing order of likelihood. We say that a prediction is correct if any one of the k locations are correct. To distinguish our variation from regular Naïve Bayes, we call this algorithm Naïve Bayes Top- k . The alteration was motivated by an

intuition that, for small values of k , a set of k predictions gives the programmer only a few places to check for the fault, with the benefit that one of them is more likely to be right. We were able to implement the alteration without modifying WEKA code.

Chapter 4.

Starting with a Correct Program

4.1 Experimental Setup

In our first experiment, we evaluate our method on a software analysis tool called JMeter [4]. There are five versions of JMeter available in the SIR, and each version passes its unit tests with no failures. We selected JMeter because it is one of the largest Java programs in the SIR. Larger programs are a more realistic usage scenario for our method, and they also pose a more difficult challenge.

We use the Naïve Bayes Top- k classification algorithm, described in Section 3.6.1, and compare the results where k is five, three, and one. We expect the rate of correct prediction to decline as k becomes smaller (see Section 3.6.1).

We also implement a trivial classifier for comparison. For a given failure, the trivial classifier predicts every location appearing in the stack trace for that failure. If any one of these locations matches the location of the fault, we say the trivial classifier correctly predicted that fault. This reflects our intuition that, if the fault’s location appears in the stack trace, it is probably easy for a programmer to find the fault by hand.

JMeter organizes its source code into modules. We run a mutation analysis on Java files in the modules “core,” “components,” and “functions.” There are two other modules we do not analyze, because they would not compile. Since these modules seem

Table 4.1: Source code statistics for JMeter

JMeter version	Java files	Total lines of code	Realistic bugs
1	232	39748	3
2	215	38579	3
3	262	46343	5
4	263	47532	1
5	268	48887	3

to be peripheral to JMeter, we ignore them.

Table 4.1 shows the number of Java files, the total number of lines of code, and the number of realistic bugs in the SIR for each of the five versions of JMeter. Actually, more bugs are provided in the SIR than are shown in Table 4.1. Some of these, however, do not cause any failures in the unit tests. Since our method requires failures, we ignore these bugs.

We applied the full set of mutation operators in Table 3.1 in the mutation analysis of each version.

4.2 Results

We say the classifier correctly predicts a fault if and only if it correctly labels at least one of that fault’s failures with the location of the fault. Note that a single fault often causes multiple failures, and this effectively gives the classifier multiple opportunities to correctly predict most faults.

Overall, the classifier is able to correctly predict the file and block location for 20.00% (3 out of 15) of the faults tested when $k = 5$. When $k = 3$, the rate drops to 13.33% (2 of 15), and when $k = 1$ it drops again to 6.67% (1 of 15). The trivial classifier correctly predicts 13.33% (2 of 15).

An alternative way of measuring prediction accuracy is to only consider whether the file is predicted correctly, and ignore block predictions. By this measure, when $k = 5$ the classifier correctly predicts 40.00% (6 of 15) of the faults tested. Again, we

see a decline in accuracy as we decrease k . When $k = 3$, the rate is 33.33% (5 of 15), and when $k = 1$ it is 26.67% (4 of 15). The trivial predictor's results are unaffected, remaining 13.33% (2 of 15).

Figure 4.1 shows a graph of these results.

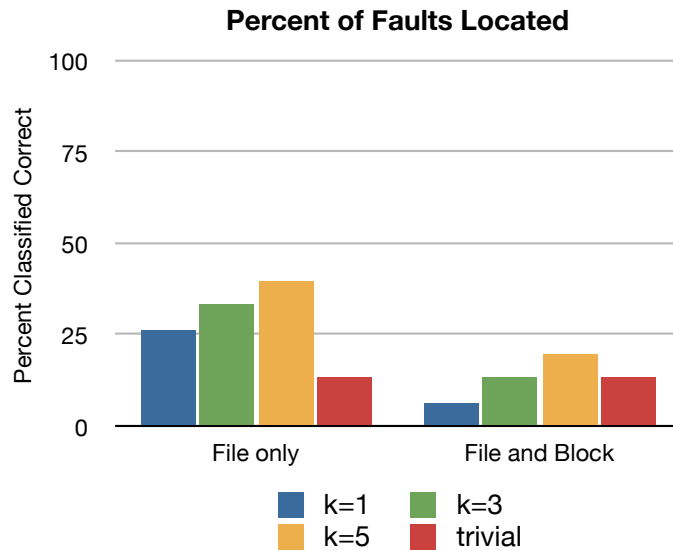


Figure 4.1: Results on JMeter using Naïve Bayes Top- k , $\forall k \in \{1, 3, 5\}$, and a trivial predictor

4.3 Evaluation

The prediction accuracy of the classifier is less than we had hoped for. There are ways to alter our method that may improve the rate of correct prediction. Those ideas are discussed in Section 8.2.

The results for our technique are mixed compared to the trivial predictor. When predicting only the file, our method outperforms the trivial predictor for all values of k . When predicting both the file and block, however, our method and the trivial predictor both classify about the same number of faults correctly.

Table 4.2: Exception classes observed in JMeter failures

Exception class	Faults observed
<code>java.lang.ArrayIndexOutOfBoundsException</code>	2
<code>java.lang.NullPointerException</code>	1
<code>java.lang.StackOverflowError</code>	1
<code>junit.framework.AssertionFailedError</code>	13

Based on the number of lines of code and the number of files in each version of JMeter, even our worst prediction rate (about 7% when $k = 1$ and predicting both the file and block) is significantly better than random guessing. For example, version five of JMeter has 268 Java files – a purely random guess as to which file contains the fault would be wrong 99.63% of the time, and randomly guessing the correct block in almost 50,000 lines of code would be even more unlikely. This comparison, however, is unrealistic. A programmer would probably have some intuition about the cause of a failure, so their guess would be much better than a random one.

We wondered whether a failure’s exception class affects its chance of correct prediction. In the failures caused by JMeter’s 15 realistic faults, we observed four different exception classes. Table 4.2 shows how many faults caused at least one failure having each exception class. The sum of the second column of Table 4.2 is greater than the number of faults observed because each fault can cause multiple failures, and each failure can have a different exception class. We can see from Table 4.2 that for three of the four exception classes, we observe only one or two faults, while thirteen of the fifteen faults induce `AssertionFailedErrors`. Figure 4.2 shows the results of the predictions using Naïve Bayes Top- k , where $k = 5$, broken down by exception class. These results indicate that accurate classification is possible for at least three of the four exception classes, but it is impossible to estimate the potential accuracy on a larger data set.

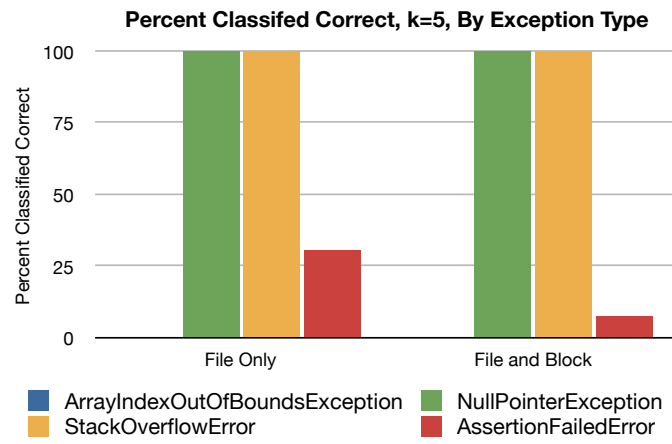


Figure 4.2: Summary of results for $k = 5$, broken down by exception

Chapter 5.

Starting with a Faulty Program

Chapter 4 evaluates the results of applying our method on a program that does not cause failures in its unit tests. In practice, however, this method is unrealistic – if we already had a non-failing program, we would have no need to locate bugs. In this chapter, we alter the experiment by analyzing mutants of a program that already contains the fault we want to locate.

5.1 Experimental Setup

The problem with mutating a faulty program is that a generated mutant will have two faults: the original one, and the fault resulting from the mutation. The presence of two faults causes the unit tests to reveal two corresponding sets of failures (usually, see below). This duplication confuses the classifier’s training algorithm – since every mutation results in a program that causes the original failures, every possible location for the original fault will be ranked as equally likely in the classifier model.

To prevent this problem, we wrote a script that strips out failures exactly matching those caused by the original fault from the mutation results. We determine if two failures match by comparing the exception class and stack trace – if both are identical, then the failures match.

Note that it is not precisely the case that every mutant will exhibit failures caused by the original fault. It is possible for a mutation to affect the original fault, causing the failures it induces to be slightly different, or to disappear entirely. In fact, we might be able to exploit this information – we discuss this possibility in Chapter 6.

We have not yet run a series of full mutation analyses on different faulty programs. Here, we describe the results of a preliminary partial analysis, designed to give an indication of whether running the full analyses is worthwhile.

In this experiment, we only mutate areas of code that are close to the original fault – within $\pm\ell$ lines of code. Then we search the results of the unit tests for examples of failures that are identical to one of the failures caused by the original fault, but were revealed in a different unit test. Such an example represents a nearly ideal observation for training the classifier, since it both closely resembles the original failure, and also correlates with a location nearby the original fault. If we can find such an example within our limited analysis, the example would also appear during the full mutation analysis. We know, therefore, that the full analysis will contain at least one observation that could lead to a correct classification. Note that the presence of such an observation is neither necessary nor sufficient for a correct prediction – we simply take it as a good sign and an indication that further investigation is justified.

We examine two Java programs from the SIR: JTopas [25] and Apache XML Security [5]. Both of these programs have three versions available, for a total of six different versions to analyze. The SIR provides a total of 71 realistic faults for these six versions. We let ℓ be 8, which means we mutate a total of 17 lines of code for each realistic fault.

5.2 Results

We found that for 16.90% (12 of 71) of the realistic faults there was at least one mutation that caused at least one failure identical to one of the original failures, but occurring in a different test case. Table 5.1 shows these results broken down by the

Table 5.1: Number of realistic bugs where the analysis revealed a failure matching an original failure, but in a different test case

Package	Realistic bugs	Bugs having property
JTopas version 1	8	0
JTopas version 2	11	0
JTopas version 3	14	5
XML Security version 1	14	3
XML Security version 2	15	4
XML Security version 3	9	0
Total	71	12

program versions tested. We were able to reveal the property in three of the six of the versions.

There were 18,383 total failures caused by our mutations, of which 102 exhibited the property we were looking for. Of those 102, 31 were `NullPointerException`s, and the other 71 were `AssertionFailedErrors`.

5.3 Evaluation

These data suggest that our method might be able to correctly predict the location of a fault in a program, even if the classifier is trained on a mutation analysis of the faulty program itself. The idea deserves further investigation.

Chapter 6.

Discussion

6.1 Mutation Analysis

At the start of this project, we knew that we wanted to use machine learning to locate faults using information about failures. The biggest problem, though, was finding a large supply of faults and failures to work with. Before settling on mutation analysis to generate the training data, we considered other options.

One idea was to use user-submitted failure reports. These are generated, for example, by programs like Firefox [18]. When Firefox crashes, it transmits information about the failure to its developers (after obtaining the user’s consent). The failure reports for Firefox are publicly available and sufficiently numerous for our purposes [30].

We rejected this idea though, partly because we were worried about how to relate these problem reports to fault locations in the program. Liblit et al [44][27][28] solve this problem, but their technique requires instrumentation of the program. We guessed that developers of any project popular enough to generate sufficient numbers of problem reports (such as Firefox) were probably averse to instrumenting their code for our research.

We also considered harvesting fault examples by relating version-control repository data with bug database information. See Section 7.2 for details on this technique.

Compared to these other techniques, though, mutation analysis seemed uniquely well-suited for our purpose. Using mutation analysis we can systematically reveal, if not all possible failures, then at least a very large number of common ones. The stack trace of a failure describes a path through the program that leads to failure under the conditions of the mutation. By analyzing the results of all the mutations, we effectively create a map that relates these failing paths with the locations of the mutations that revealed them. Our assumption is that, often, a set of similar failing paths through the program will be strongly associated with a small number of particular mutation locations. This assumption reflects an intuition that many sets of similar failures have “hot” points in the code, that is, places where there are many ways for a programmer to mistakenly cause that failure to occur.

Mutation analysis, however, has one serious downside – we have no way to know how representative our mutations are of real program failures. It is certain that they are not completely representative, since there are innumerable types of bugs that we do not model with mutation operators. In addition, it is far from clear that our mutations cover all of the most common programming bugs. And, even if they did cover the most common bugs, it is probably not the case that they cover them in the correct proportions – we would expect some bugs to be more common than others, and even to vary from project to project and programmer to programmer. This issue is probably the most serious limitation to the potential accuracy of our method, and may explain why the results in Chapter 4 did not meet our expectations. Unless our mutation analysis correctly models the distribution of bugs, our classifier is, in a sense, predicting the wrong thing.

6.2 Machine Learning

The idea to use a machine learning-based classification algorithm was an important step in the development of this project. Machine learning enables us to exploit the amount of data we generate through mutation analysis.

Machine learning is generally considered a branch of artificial intelligence, but the “intelligence” is most often based on applied statistics [13]. In this project, as in many statistical applications, we want to use previously observed data to extrapolate information about new data. One goal of machine learning is making the process of extrapolation computationally efficient, and due to our large data set, this is also important to us. More traditional techniques, such as decision trees, would have been much less efficient, since our data has many different possible values for each variable [13].

There are many powerful and efficient machine learning classification algorithms besides Naïve Bayes [13]. It is possible that use of another algorithm would have improved our classification accuracy. We suspect, though, as discussed in Section 6.1, that a more serious limitation is the quality of the data from the mutation analysis. Since machine learning classifiers extrapolate from training data, their accuracy is highly dependent on the quality of that data.

As an matter of programming technique, we found it helpful when testing our machine learning algorithm to introduce simple sanity checks. For example, we implemented a check showing a degradation in the accuracy of classification as the size of the training data set was reduced. If the classifier’s accuracy remained constant, or improved, it would mean something is wrong with the data, the algorithm, or both.

6.3 Source Code Locations

In this project, we spent a lot of time working with representations of locations in source code. We wanted a useful way to specify the place that a fault could be found in a set of source files.

Our first location representation scheme used a file and line number pair that identified the first line in the file where the fault began. It quickly became clear that this was insufficient for our needs. Single line numbers are too precise. Since mutations are injected at almost every line in the program, single line numbers provide too many potential locations to predict accurately.

The next scheme, and the one we use in this thesis, represents a location with a file and two line numbers that specify the start and end of a code block. Code blocks are defined in Java by matching pairs of braces. An informal examination of the source code in the SIR found that code blocks are typically only around 10 to 20 lines long – short enough to be useful as a location. Using this scheme, the classifier’s rate of prediction improved. This stands to reason, since a smaller set of locations means fewer choices for the classifier to choose between.

We found a problem, however, with using the file/block format. The mutator loads Java files and, after modifying the AST, prints them out again with a significantly different mapping between line numbers and code. For example, comments are removed, and non-standard whitespace is used in the formatting. This means that locations in a mutant do not match locations in the original source file. We address this problem as described in Section 3.2.4, but found that there were still discrepancies between locations. This can occur, for example, if a two-line mutation replaces a one-line statement in the original file – every location after that mutation will be off by one.

This highlights a more general problem with representing locations using line numbers: line numbers do not strictly correspond to statements. We would like to be able to specify the location of a block or statement in a source file in a more direct way, so that small alterations to the file do not invalidate those locations. We have not yet found a satisfactory way to address this problem.

6.4 Synthetic vs Real Faults

As noted in Section 6.1, one of the key features of our method is the ability to automatically generate faults and failures for the training data set. We faced a problem, though, in finding realistic bugs for evaluating the classifier’s accuracy.

Developers of machine learning algorithms will often divide their data into two parts – some examples are used for training, and the rest for testing. We decided that we could not use mutations as test data. We are not really interested in evaluating the

learning algorithm (the performance of Naïve Bayes is already well documented [15]), but rather in evaluating the quality of training data from a mutation analysis. Testing on mutation data would inflate our results, and we would not be able to expect the same performance on real bugs.

So, we needed real bugs to test the classifier. One option was to mine the bug tracking systems of open-source projects (see Section 7.2). We rejected this option, because our method can only locate bugs that cause unit test failures, and we reasoned that bugs mined from bug tracking systems may often not cause those failures. This is because, for a bug to be in a bug tracking system, that bug must have been committed to the project's source code version control system. This implies that the bug probably did not cause the unit tests to fail. If it had, the programmer would have noticed and fixed the bug before committing the faulty code to version control.

We tried another approach to collecting real bugs, where we wrote a plug-in for the Eclipse IDE [14] that would remind the user to send us exceptions along with the fixed code that resolved the problem. Although we distributed it widely among graduate students using Eclipse at UCSD, we did not receive a sufficient number of responses.

In the end, we feel fortunate to have found the SIR database containing both organized versions of source code and realistic bugs. The disadvantage to using bugs from the SIR is that they are not real. That is, they were created after the program's development by researchers, not by the developers themselves [12]. We do not know, therefore, to what extent our test bugs resemble real bugs that would be encountered by a developer as they are in the midst of development.

It is worth noting that we were careful not to examine the bugs we were testing until after we had used them. If we had not done this, we would have risked biasing the development of our mutation operators to cause mutations similar to the test bugs. While we wanted mutations to be similar to real bugs in general, we could not have them specifically tailored to resemble the test set.

Chapter 7.

Related Work

7.1 Locating Faults

There are many other ways to locate faults in source code. This section presents an overview of some of these techniques.

7.1.1 Slicing

Slicing [41] is a method for exploring source code, but it is also frequently used for locating faults [40]. It is a kind of static analysis that looks “backwards” from a seed statement to find all the other statements in the program that affect the seed. This can be useful in large programs, especially for debugging. For example, a programmer may know that a variable is being assigned the wrong value, but does not know where the faulty assignment is located, or even which parts of the program affect that variable. With slicing, the programmer can find all the statements that affects the value of the variable up to that point in the program. Variants of this method, such as Thin Slicing [38], reduce the number of statements to be examined even further. Slicing is a powerful technique, but for locating bugs it has a serious deficiency: a programmer must already know the location of an erroneous statement before he can slice it and find the fault.

7.1.2 Change Classification

Another approach to locating bugs in the code is a static analysis of the dynamics of test cases [39]. Tools like Chianti [35] take as input two versions of a program: the most current one, which is failing at least one test case, and a previous version that passes all its test cases. The idea is that the failing test cases must be caused by changes that have been made between the correct version and the faulty one. Additionally, Chianti can map which sections of recently changed code are called by which test cases, and then, based on which test cases are failing, identify locations in the code that are likely to be causing the failure. This technique relies on a clever way of finding the “atomic” differences between two versions of a program. We think that combining this technique with our own might be very powerful – Chianti could find a small set of possible locations for a bug, and then our method would narrow this set down even further.

7.1.3 Delta Debugging

Delta Debugging [42] [43] is another technique for locating faults from information about failures. Delta Debugging takes a failing program and generates a large series of different test cases designed to isolate the conditions that cause the failure. By testing the failing program with these cases, Delta Debugging narrows down the set of dynamic program states that lead to a failure. Hopefully, at the end of this process, the tool reveals some small set of program states that lead to failure. An example outcome might be “The program fails if and only if variable x gets assigned value y .” This information can then lead the programmer to the fault. Delta Debugging is unique in that it looks at the differences between the running program’s states. Like our technique, it does not require instrumentation or even prior knowledge of the source code. Also like our technique, it is computationally expensive and time-consuming.

7.1.4 Program Invariants

Another method for finding bugs is use of invariants. Invariants are properties of executing programs that should not be violated during normal (that is, correct) execution [17]. For example, consider a loop that is supposed to iterate a variable from zero to five and then exit. We can infer at least two invariants in this case: first, the variable should never be less than zero, and second, the variable should never be more than five. If either of these conditions is violated during the program's execution, it means there is probably a fault in the program. In many cases knowing which invariant was violated can direct the programmer to the location of the fault. Continuing our example, if the loop variable is assigned a value of six (and thus the invariant violated), the programmer might look at the statements and conditions where that variable is incremented.

Once a programmer knows a program's invariants, he can instrument the code to detect and report violations. The problem is how to derive the invariants in the first place.

Specifications

One way to discover the invariants of a program is to have the programmer explicitly describe them in a specification [11] [16]. This specification is parsed by the invariant checker, which then reports violations. This technique has a serious downside in that writing specifications is tedious and time-consuming. Worse, the programmer is often unaware of subtle or complex invariants that are often involved in the most difficult bugs.

Dynamic Invariant Detection

A different and arguably better way of inferring invariants is to discover them automatically. One example of a tool that uses this method is Daikon [17]. Daikon starts by assuming a liberal set of invariants for a given program, and then gradually rules some out by running the program through many test cases and eliminating invariants

that do not hold. Those that are left over after the elimination process are likely to be real program invariants. This approach is automatic, and so is not subject to human error in the specifications themselves. It is also pervasive, in the sense that it finds invariants in the whole program. In contrast, when programmers write specifications they tend to focus on the areas that they think will be problematic, and they are not always right.

Another approach is taken by a tool called DIDUCE [23]. DIDUCE works like Daikon, but continuously checks the posited invariants against the state of current program execution. As the invariants are violated, they are eliminated from the set of posited invariants, and the change is reported to the operator. In this way, the programmer can go back and look at the DIDUCE log of a long-running program and see the last set of invariants that were violated immediately before the program crashed.

7.1.5 Machine Learning on Program Executions

Brun and Ernst have researched a technique that uses machine learning to rank properties of source code by the likelihood that a given property causes a fault [8]. Their implementation uses dynamic program properties, but they point out that it could be used with static properties as well. The idea is to train a classifier on properties of faulty programs, and also on correct versions of those programs. The programmer then uses the trained model to classify properties of a new program by how likely they are to reveal faults. The programmer can then focus his attention on those properties. Their implementation, called the Fault Invariant Classifier, automatically determines program properties using dynamic analysis (Daikon, in fact, see section 7.1.4), so the technique does not require special instrumentation or programmer-written specifications.

This technique is similar to our own in spirit, but differs since the machine learning models are different. Our model is trained on the characteristics of failures, rather than on general properties of failing programs. As such, our model is only applicable to the versions of the program we trained it on, while theirs extends to other programs as well.

7.2 Bugs for Testing

To test our classifier, we need examples of bugs from real world projects. Eventually we settled on using the realistic bugs in the SIR (see section 2.2.1) because they are very convenient. There were alternatives, however.

One idea was to collect the bugs ourselves from graduate student programmers at UCSD. As discussed in section 6.4, this approach did not work very well.

Another idea is to look for bugs in the historical artifacts of a software project [10][19][37]. For example, Zeller et al (in [37]) demonstrate a method that relates changes checked into a CVS [26] code repository with bugs tracked using Bugzilla [9]. Their tool analyzes a project's CVS logs, looking for examples of check-in messages that contain Bugzilla tracking numbers and state that they fix the bug. This process links bug reports in Bugzilla with their fixes in CVS. The tool then uses the project version number from when the bug was reported to identify the version of the code where the fix can be applied. This method has been successfully demonstrated on large projects like Eclipse.

Chapter 8.

Conclusion

8.1 Summary of Results

In this research, we found that it is sometimes possible to locate a fault from the failures it causes. This is done by training a machine learning classifier with fault and failure data obtained from a mutation analysis of a correct version of the program. We also found evidence justifying further investigation into a variation of the technique that does not require a correct program.

In general, we are disappointed with the results of our experiments. Our classifier was only able to locate a fault to within the correct source file 40% of the time. That is significantly better than random guessing, but not much better than a trivial classifier. In any event, a tool that is only right 40% of the time would not be very useful to developers. In Section 8.2, we outline some ideas that may increase the accuracy of our method.

8.2 Future Work

8.2.1 Improving Accuracy

One idea to improve the accuracy of our method would be to reduce the set of possible fault locations that we predict. Currently, we assume the bug could be anywhere in the program. This assumption may be unnecessarily conservative. For example, we may be able to discover what areas of code have been recently changed (see Section 7.1.2 for one idea on how to do this). We also might be able to use information that indicates which regions of a program have historically contained more bugs (perhaps by mining a bug tracking system – see Section 7.2) to adjust the confidence levels of our predictions.

8.2.2 Other Work

We intend to complete our investigation of training using a mutation analysis of a faulty program. We would like to know the overall rate of correct prediction in this case, and compare this with the results of training on a correct program.

We would also like to try our method on different programs. Currently, our experimental results training on data from a correct program are based on only one program (JMeter). We might find that our method performs more or less accurately on other programs, depending on the characteristics of the code.

Mutation analysis is quite slow. Currently, a complete mutation analysis of JMeter takes approximately 8-10 hours on FWGrid – far from adequate for a developer to use interactively. We would like to explore ways to speed it up. One idea is to cache mutation results, reusing them if it can be shown through static analysis that they will not be affected by recent changes to the code.

References

- [1] AGRAWAL, H., DEMILLO, R. A., HATHAWAY, R., HSU, W., HSU, W., KRAUSER, E. W., MARTIN, R. J., MATHUR, A. P., AND SPAFFORD, E. H. Design of mutant operators for the C programming language. *Technical Report SERC-TR-41-P* (1989).
- [2] Apache Ant. <http://ant.apache.org/>.
- [3] Apache Ant user manual. <http://ant.apache.org/manual/index.html>.
- [4] Apache JMeter. <http://jakarta.apache.org/jmeter/>.
- [5] Apache XML security. <http://santuario.apache.org/>.
- [6] BIEMAN, J. M., GHOSH, S., AND ALEXANDER, R. T. A technique for mutation of java objects. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering* (Washington, DC, USA, 2001), IEEE Computer Society, p. 337.
- [7] BISHOP, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*, first ed. Springer, August 2006.
- [8] BRUN, Y., AND ERNST, M. D. Finding latent code errors via machine learning over program executions. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 480–490.
- [9] Bugzilla home. <http://www.bugzilla.org/>.
- [10] CUBRANIC, D., SINGER, J., AND BOOTH, K. S. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.* 31, 6 (2005), 446–465. Member-Gail C. Murphy.
- [11] DELINE, R., AND FÄHNDRICH, M. Enforcing high-level protocols in low-level software. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), ACM Press, pp. 59–69.

- [12] DO, H., ELBAUM, S. G., AND ROTHERMEL, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal* 10, 4 (2005), 405–435.
- [13] DUDA, R. O., HART, P. E., AND STORK, D. G. *Pattern Classification*. Wiley-Interscience Publication, 2000.
- [14] Eclipse. <http://www.eclipse.org/>.
- [15] ELKAN, C. Boosting and naive bayesian learning, 1997.
- [16] ENGLER, D., CHELF, B., CHOU, A., AND HALLEM, S. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation* (Berkeley, CA, USA, 2000), USENIX Association, pp. 1–1.
- [17] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (Los Alamitos, CA, USA, 1999), IEEE Computer Society Press, pp. 213–224.
- [18] Firefox. <http://en.www.mozilla.com/en/firefox/>.
- [19] FISCHER, M., PINZGER, M., AND GALL, H. Analyzing and relating bug report data for feature tracking. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering* (Washington, DC, USA, 2003), IEEE Computer Society, p. 90.
- [20] FRANK, E., TRIGG, L., HOLMES, G., AND WITTEN, I. H. Technical note: Naive bayes for regression. *Mach. Learn.* 41, 1 (2000), 5–25.
- [21] FWGrid project. <http://fwgrid.ucsd.edu/>.
- [22] GAMMA, E., HELM, R., JOHNSON, R. E., AND VLISSIDES, J. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, March 1995. ISBN-10: 0201633612 ISBN-13: 978-0201633610.
- [23] HANGAL, S., AND LAM, M. S. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ACM Press, pp. 291–301.
- [24] HSUEH, M.-C., TSAI, T. K., AND IYER, R. K. Fault injection techniques and tools. *Computer* 30, 4 (1997), 75–82.
- [25] JTopas - Java tokenizer and parser tools. <http://jtopas.sourceforge.net/jtopas/>.
- [26] KRAUSE, R. CVS: an introduction. *Linux J.* 2001, 87 (2001), 3.

- [27] LIBLIT, B., AIKEN, A., ZHENG, A. X., AND JORDAN, M. I. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, June 9–11 2003).
- [28] LIBLIT, B. R. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, dec 2004.
- [29] Gnu make. <http://www.gnu.org/software/make/>.
- [30] Mozilla talkback. <http://talkback-public.mozilla.org/>.
- [31] NYSTROM, N., CLARKSON, M., AND MYERS, A. Polyglot: An extensible compiler framework for java, 2003.
- [32] OFFUTT, A. J., LEE, A., ROTHERMEL, G., UNTCH, R. H., AND ZAPF, C. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.* 5, 2 (1996), 99–118.
- [33] How to use polyglot.
<http://www.cs.cornell.edu/projects/polyglot/Overview.html>.
- [34] Polyglot extensible compiler framework.
<http://www.cs.cornell.edu/projects/polyglot/>.
- [35] REN, X., SHAH, F., TIP, F., RYDER, B. G., AND CHESLEY, O. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Not.* 39, 10 (2004), 432–448.
- [36] Sun grid engine. <http://gridengine.sunsource.net/>.
- [37] ŚLIWERSKI, J., ZIMMERMANN, T., AND ZELLER, A. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes* 30, 4 (2005), 1–5.
- [38] SRIDHARAN, M., FINK, S. J., AND BODIK, R. Thin slicing. *SIGPLAN Not.* 42, 6 (2007), 112–122.
- [39] STOERZER, M., RYDER, B. G., REN, X., AND TIP, F. Finding failure-inducing changes in java programs using change classification. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2006), ACM Press, pp. 57–68.
- [40] TIP, F. A survey of program slicing techniques. Tech. rep., Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, The Netherlands, 1994.
- [41] WEISER, M. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering* (Piscataway, NJ, USA, 1981), IEEE Press, pp. 439–449.

- [42] ZELLER, A. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering* (New York, NY, USA, 2002), ACM Press, pp. 1–10.
- [43] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.* 28, 2 (2002), 183–200.
- [44] ZHENG, A. X., JORDAN, M. I., LIBLIT, B., AND AIKEN, A. Statistical debugging of sampled programs. In *Advances in Neural Information Processing Systems 16*, S. Thrun, L. Saul, and B. Schölkopf, Eds. MIT Press, Cambridge, MA, 2004.