2014

A Fault-Based Model of Fault Localization Techniques

Mark A. Hays
University of Kentucky, mark.hays@uky.edu

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Mark A. Hays, Student

Dr. Jane Huffman Hayes, Major Professor

Dr. Miroslaw Truszczynski, Director of Graduate Studies

A FAULT-BASED MODEL OF FAULT LOCALIZATION TECHNIQUES

DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in the
College of Engineering
at the University of Kentucky

By
Mark Allen Hays
Lexington, Kentucky

Director: Dr. Jane Huffman Hayes, Professor of Computer Science
Lexington, Kentucky

2014

ABSTRACT OF DISSERTATION

A FAULT-BASED MODEL OF FAULT LOCALIZATION TECHNIQUES

Every day, ordinary people depend on software working properly. We take it for granted; from banking software, to railroad switching software, to flight control software, to software that controls medical devices such as pacemakers or even gas pumps, our lives are touched by software that we expect to work. It is well known that the main technique/activity used to ensure the quality of software is testing. Often it is the only quality assurance activity undertaken, making it that much more important.

In a typical experiment studying these techniques, a researcher will intentionally *seed a fault* (intentionally breaking the functionality of some source code) with the hopes that the automated techniques under study will be able to identify the fault's location in the source code. These faults are picked arbitrarily; there is potential for bias in the selection of the faults. Previous researchers have established an ontology for understanding or expressing this bias called *fault size*. This research captures the fault size ontology in the form of a probabilistic model. The results of applying this model to measure fault size suggest that many faults generated through *program mutation* (the systematic replacement of source code operators to create faults) are very large and easily found. Secondary measures generated in the assessment of the model suggest a new static analysis method, called *testability*, for predicting the likelihood that code will contain a fault in the future.

While software testing researchers are not statisticians, they nonetheless make extensive use of statistics in their experiments to assess fault localization techniques. Researchers often select their statistical techniques without justification. This is a very worrisome situation because it can lead to incorrect conclusions about the significance of research. This research introduces an algorithm, MeansTest, which helps automate some aspects of the selection of appropriate statistical techniques. The results of an evaluation of MeansTest suggest that MeansTest performs well relative to its peers. This research then surveys recent work in software testing using MeansTest to evaluate the significance of researchers' work. The results of the survey indicate that software testing researchers are underreporting the significance of their work.

KEYWORDS: Software testing, testability, statistical analysis, MeansTest, empirical validation

Mark Hays
Student's signature

04/29/2014
Date

A FAULT-BASED MODEL OF

FAULT LOCALIZATION TECHNIQUES

By

Mark Allen Hays

Dr. Jane Huffman Hayes
Director of Dissertation

Dr. Miroslaw Truszczynski
Director of Graduate Studies

04/29/2014
Date

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Hayes, as well as my committee, (consisting of Dr. Stromberg, Dr. Liu, and Dr. Truszczynski) for their service in my committee and assistance in my research. I would also like to thank Dr. Bathke for his assistance with my MeansTest research.

I would like to thank Nan Li and Dr. Jeff Offutt from George Mason University for providing the latest version of their Graph Coverage tool.

This work was funded in part by the National Science Foundation under grant CCF-0811140 (research) and ARRA-MRI-R2 500733SG067 (benchmark development for Tracelab).

Table of Contents

List of Tables

List of Figures

Chapter 1. General Introduction

Software is expensive to build and maintain at a high quality. According to the Department of Commerce, activities specifically geared toward improving software quality account for 50% of a typical software product's budget, altogether costing "$59.5 billion annually [1]." Quality takes many forms. Tangible aspects of quality include correct functionality and precise performance constraints. Intangible aspects include user-friendliness and security. While high quality of this nature may seem abstract, low quality has a very tangible impact. For instance, the healthcare.gov Web site suffered a catastrophic failure to launch on October 1st, 2013 that lasted two months. During this period, the White House recruited numerous consultants to fix the site, more than doubling the estimated cost of developing the site from the original $400 million as of October 1st to $1 billion dollars as of December 4th [2].

According to the IEEE Software Engineering Body of Knowledge, most quality assurance activities that students learn to apply take the form of software testing [3]. *Software testing* is the process of actively seeking and eliminating *software bugs*. As mentioned above, software testing is a big business; some consultants, such as Galmont Consulting, profit exclusively through their testing services. Larger companies, such as Google and IBM, maintain internal software quality assurance operations, but also enter this market through their "global services" consultancy divisions. Other companies enter the market indirectly by providing tool support for testing. For instance, IBM sells the Rational Functional Tester to help testers write test cases for varied graphical user interfaces. Given the unprecedented scope of the healthcare.gov project and the high political stakes at the time, these companies that were contracted to fix the site had much to gain financially by selling their latest and greatest testing services. All software testing companies have an ethical obligation to provide testing services backed by testing theory that has been subjected to empirical validation.

Researchers must fulfill their part of this ethical obligation: empirical validation must emulate the types of quality issues that arise in industry. Researchers have several options to provide this emulation. For instance, researchers sometimes back-test new theory against old bugs found in older versions of open-source software. Another popular approach is to introduce many small, random changes to the software, called *program mutation*. While this practice does not sound realistic, it appeals to some researchers because it is a low cost way to construct experiments. Regardless of the approach taken, researchers currently have no way of knowing whether the

bugs they select are representative of the bugs that arise in industry. It could be the case that researchers are inadvertently selecting bugs that are easy for their tools to find.

In this respect, researchers also have an ethical obligation to prove that new theory works beyond reasonable doubt. Researchers must acknowledge the possibility that a new testing theory could eliminate a randomly selected set of bugs by chance, but not most bugs in practice. Researchers have been trying to design standard templates for software testing experiments that allow other researchers to determine the *statistical significance* of their work. Statistical *hypothesis tests* are essential to this end because they allow researchers to quantify the true value of their work. However, the correct construction of experiments is a highly nuanced process that requires the researcher possess considerable statistical background knowledge. Unfortunately, this research finds that it is usually not the case that experiments are properly constructed.

Put succinctly, this dissertation is "research-of-research" that improves the state of the art of research into new software testing theory. It provides a new quantitative way of comparing faults, called *fault size*, that directly measures difficulty. It revisits the way statistical analysis is performed and imparts a new algorithm, MeansTest, for automating some parts of the analysis that are being skipped by researchers due to complexity.

1.1. Problem Statements

Experimentation in software testing research focuses on determining the location of a bug given one or more failing inputs to the software, called *test cases*. The practice of locating a bug is known as *fault localization*. Some fault localization techniques are said to be *spectrum-based* in that they use data about the execution of code during failing test cases to predict where the fault occurs.

Unfortunately, there is limited theoretical foundation for understanding why spectrum-based methods work. The quality of spectrum-based fault localization is often described in the language of information retrieval quality metrics. While the theory provides no possibility that spectrum-based fault localization could be wrong, all experiments show non-negligible erroneous classifications. Thus a major problem this research addresses is revising the existing model of spectrum-based fault localization to explain the erroneous classifications.

To make matters worse, these researchers are often not statisticians and may not have access to a member of the statistics department, so their statistical conclusions may also be suspect. Within

the broader software engineering field, there is a discussion every few years on how software testing experiments should be performed. Recently, there was a paper [4] which concluded that all researchers should use a particular hypothesis test, the Mann-Whitney U test, to assess all experiments involving randomized optimization algorithms. This result has been oversimplified within the software testing community and now many researchers use this test, even when they are not studying randomized optimization algorithms. An anonymous reviewer on one of the papers in this dissertation anecdotally suggests that this trend has arisen out of laziness:

> As the authors point out, we are not statisticians and generally don't have the luxury of sending our data to one. As researchers, we often choose tests to employ because they're either safe or simply what we always use. I, the anonymous reviewer, am guilty of one of the crimes the authors mention - I almost always employ a conservative Mann-Whitney U test to evaluate my work because I do not want to make distributional assumptions.
>
> Anonymous ICST 2014 Reviewer

This research seeks to address these issues by introducing and evaluating a more comprehensive approach to statistical analysis of software testing experiments that is partially automated.

1.2. Thesis

The thesis of this research is: software testing experimentation can be improved through a better understanding of experiment design. A quantitative model of *fault size* can show how *software testability* of code masks the true *intrinsic size* of faults. A workflow for a *typical statistical analysis* can help automate some aspects of the evaluation of software testing techniques.

1.3. Contributions

This dissertation makes several contributions. The software testability measure, itself, is an improvement over the state of the art, McCabe's cyclomatic complexity. This improvement is demonstrated through empirical validation on very large and well-known software, in which the recall, precision, and F1 of the methods are compared. The study found that the testability measure had higher recall than McCabe's version. The model of fault size has been applied to a set of well-known faults and demonstrates that these faults are very easy to find, raising concerns about the quality of experimentation in this area. The workflow of statistical analysis has been published in the form of an automated tool in several forms and is available for public use. The

dissertation contributes an empirical validation of the workflow, demonstrating that it is more effective than naively relying on any single statistical hypothesis test. The practical benefits of the workflow are demonstrated through a meta-analysis of existing software testing research. A more complete description of contributions is mentioned in context in Chapter 6: Conclusions and Main Contributions.

1.4. Organization

Chapter 2 presents the literature review. Chapter 3 summarizes the techniques developed. Chapter 4 presents a synthesis of the dissertation research. Chapter 5 presents future work. Chapter 6 presents conclusions and main contributions. The subsequent chapters are organized as a compilation of published research. Chapters 6 and 7 outline software testability and the model of fault size. Chapters 8 and 9 validate the workflow for statistical analysis.

Chapter 2. Literature review

This section discusses the problems faced in software testing experiments and related work.

2.1. Definitions

2.1.1. **Testing**

Testing is defined as "an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component [5]." The goal of software testing in this research is to find bugs in software.

Testing involves constructing *test cases*. A test case describes the expected behavior of a program. Typically a test case invokes the program under test with a specific set of input values and verifies that the output is correct. The size of the program's *domain* (the possible inputs to the program) determines the potential size of the universe of test cases. In general, this universe is too large to entirely test, so the science of software testing is selecting test cases effectively.

A complete set of test cases is called a *test set*. In practice, a program has exactly one test set: the one the testers wrote. In terms of set theory, the universe of test sets for a given program is precisely the Kleene closure on the universe of test cases. Researchers in software testing create heuristics for selecting "useful" test sets from this universe; these heuristics are called *coverage criteria*. A family of coverage criteria of interest to this research is *structural testing*. Structural testing is defined as "testing that takes into account the internal mechanism of a system or component [6]."

2.1.2. **Control flow graphs and blocks**

In structural testing, programs are represented as *control flow graphs*. A control flow graph is "a diagram that depicts the set of all possible sequences in which operations may be performed during the execution of a system or program [6]." In this research, the term control flow graph refers to the flowchart of a single function's execution.

The nodes in the graph represent *blocks*. A block is "a group of contiguous storage locations, computer program statements, records, words, characters, or bits that are treated as a unit [6]." Formally, let $e(s)$ be the event that statement s is executed. A block b is a given set of statements that satisfy the following theorem:

$$\exists s(s \in b \rightarrow e(s)) \rightarrow \forall s(s \in b \rightarrow e(s)) \quad (1)$$

The theorem reads: if one statement in a block is executed, then all statements in the same block are also executed.

The edges in the graph represent jump statements, such as if-else statements and while loops. A jump statement always ends the block containing the jump. The destination of the jump marks a new block; there is an edge from the jump to its destination in the control flow graph. In the case of conditional jumps, which simply go to the next statement when the jump's condition is false, the next statement implicitly marks the start of a new block and so there is always an implicit edge between a conditional jump and the next statement.

Listing 2.1 gives a sample function and Figure 2.1 represents its control flow graph.

```
public static List<CFG> parseCFGs(String fileName){ // B11
      // B0
      List<CFG> list=new LinkedList<CFG>();
      CompilationModelImpl cm=new CompilationModelImpl();
      cm.parseInputFile(fileName);
      List<Compilation> clist=cm.getCompilations();
      for(Compilation c : clist){ // B1
            // B2
            for(CompilationElement e : c.getElements()){ // B4
                  if(e instanceof ControlFlowGraph // B5
                  && ((ControlFlowGraph)e).hasHir() // B7
                  &&e.getName().contains("HIR")){ // B9
                        // B10
                        List<at.ssw.visualizer.model.cfg.BasicBlock>
                              blist=((ControlFlowGraph)e)
                              .getBasicBlocks();
                        CFG g=new CFG(c.getMethod(), blist.toArray(new
                              at.ssw.visualizer.
                              model.cfg.BasicBlock[0]));
                        list.add(g);
                  }
                  // B8
            }
            // B6
      }
      // B3
      return list;
}
```

Listing 2.1. parseCFGs function

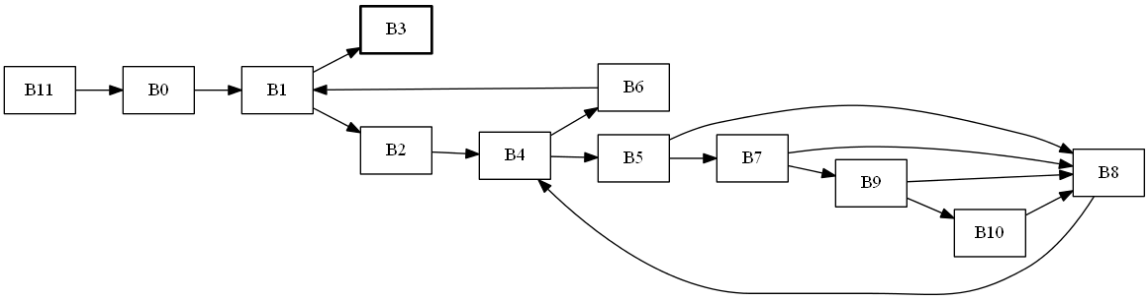

Figure 2.1. Control flow graph for parseCFGs

In the graph, B11 represents the entry point to the function and B3 represents the exit point. There are two loops in the function and the graph indicates them.

2.1.3. **Coverage**

Code coverage is defined as "a measure of the occurrence of certain behavior during (typically dynamic) functional verification and, therefore, a measure of the completeness of the (dynamic) functional verification process [7]." Code coverage provides a stopping point for structural testing. There are many levels of coverage. The most basic level is called *node coverage* or *statement coverage* and requires that the test set execute all nodes in the graph at least once. In the above graph, a tester would have to execute all nodes in {B0, B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11} at least once by executing the function with one or more test cases. More advanced forms of coverage exist, for example *edge coverage*, which would require executing all of the edges in the graph: {[B11,B0], [B0,B1], [B1,B3], [B1,B2], [B2,B4], [B4,B6], [B4,B5], [B5,B8], [B5,B7], [B7,B8], [B7,B9], [B9,B8], [B9,B10], [B10,B8], [B8,B4], [B6,B1]}. The most complicated level is called *path coverage* and requires that test cases execute every single possible path through the graph. It is actually not possible to enumerate the path coverage requirements for the above graph; for programs containing loops, path coverage is not feasible [8].

2.1.4. **Faults, failures, fault seeding**

The goal of software testing activities, such as structural testing, is to find *faults*. A fault is "an incorrect step, process, or data definition in a computer program [9]." The manifestation of a fault on the behavior of a program is called a *failure*. A failure is defined as "the inability of a system or component to perform its required functions within specified performance requirements [9]." It is desirable to find faults, through activities such as testing, before the customer receives the software.

To demonstrate that a test set can find faults, researchers implement a process called *fault seeding*. Fault seeding is defined as "the process of intentionally adding known faults to those already in a computer program for the purpose of monitoring the rate of detection and removal, and estimating the number of faults remaining in the program [6]." In software testing research, a common goal is to show that a particular structural testing method is more effective at removing faults than some baseline method. Rather than studying the method in a longitudinal study of a real software project, the researcher will seed old faults back into existing software to simulate the method's effectiveness. In the absence of historical data, researchers will use program mutation to generate simple faults; for instance, by switching operators.

2.1.5. **Confusion matrix, recall, and precision**

In information retrieval, a list retrieved in response to a query is evaluated for accuracy. Typically, the analyst will somehow classify the data into true positives, false positives, true negatives, and false negatives. These classifications form a 2x2 *confusion matrix*. Table 2.1 gives an example confusion matrix.

Table 2.1. Observed confusion matrix

	TRUE	**FALSE**
Positive	11	4557
Negative	18081	330

As Table 2.1 shows, the query correctly returned 11 elements; these are called *true positives*. It correctly ignored 18081 elements; these are called *true negatives*. It incorrectly returned 4557 elements; these are called *false positives*. It missed 330 elements; these are called *false negatives*. A number of metrics exist to measure the quality of a confusion matrix. *Recall* is the percent of recovered true positives:

$$\text{recall} = \frac{\text{TP}}{\text{TP+FN}}. \quad (1)$$

It is trivial to get 100% recall because returning the entire data set will guarantee 100% recall [9]. Thus researchers vie to achieve as high a recall as possible using only a minimal amount of data. In software testing research, recall is known synonymously as *Defect Detection Effectiveness*.

Precision is the percent of correct retrievals:

$$\text{precision} = \frac{\text{TP}}{\text{TP+FP}}. \quad (2)$$

It is non-trivial to get 100% precision [9]. In fault proneness research, a common threat to validity with measuring precision is that the absence of a fault in a component's historical data does not rule out the possibility of a fault appearing in the future; the false positive rate could be overstated. Nonetheless, precision continues to see use in fault proneness research.

2.1.6. **Systematic error**

Systematic error is defined as "the portion of error that is repeatable, i.e., zero error, gain or scale error, and linearity error [10]." If a researcher does not notice that a systematic error is present, the error could bias the experiment results. An example of a systematic error would be seeding only "easy" faults that a test set is guaranteed to catch. This error would result in a gain on the Defect Detection Effectiveness that would be erroneously attributed to the quality of the test set.

2.2. Software Testing Experiments

This section describes how typical software testing experiments are constructed. To be clear, this research does not employ typical software testing experiments. Instead, this section highlights issues with software testing experiments that motivate this research.

Frankl and Weiss described a typical software testing experiment where they compared the effectiveness of two structured testing criteria in a statistically significant way. The criteria they compared were *all-edges* (the test set must execute all edges in the control flow graph at least once) and *all-uses* (for each variable defined in the program, the test set must execute all uses of the variable at least once). Frankl and Weiss picked nine programs to test. They described, for each program, the "universe" of test inputs they created. They assumed that the failure rates would be normal except in cases where all test sets always found all bugs. Frankl and Weiss generated ~5000 test sets for both criteria. For each test set, they used a random algorithm to add test cases to the set, followed by greedy tweaking to add and remove test cases until the desired coverage level was reached. Using this method, they produced at least 30 test sets per 2% coverage interval for each program. They created a range of faulty versions for each program. Then they ran the tests against the faulty programs and reported how many test sets reported a fault as a fault detection ratio. To test their hypothesis that all-uses found more bugs than all-edges, they computed the confidence interval around all-edges and rejected the null hypothesis

when the mean bug find rate of all-uses fell outside the interval [10]. This paper is included here because it is representative of typical testing experiments that our research seeks to improve.

Dit et al. [11] surveyed software maintenance papers from the last ten years. Their goal was to reproduce the tools and results of these papers in the TraceLab framework, then make the components publicly available. They organized the papers using the Petersen et al. systematic mapping process [12]. Their steps include: 1) Defining the research questions; 2) Conducting the search; 3) Selecting screening criteria; 4) Classifying the technique (in their case: determining the tracers and preprocessors used); and 5) Extracting the data. This research reuses the mapping process of Dit et al.

Basili et al. surveyed empirical software engineering papers [13]. They created a framework that describes experiments in terms of its definition, design, implementation, and interpretation. Each part of the framework has several attributes, such as scope (single project, replicated project, multi-project variation, blocked subject-project), perspective, and impact. The authors used this framework to systematically survey research papers. They identified several problem areas: 1) that there is no consistency among practitioners, 2) that experiments are hard to precisely define because there is no standard metric for gauging software quality, 3) the experiment plan should contain ideas for subsequent experiments, 4) experiments need to be published in a repeatable and extendable way, and 5) results need to be qualified by the controlled variables [13]. Whereas Basili et al. provide a general experimentation framework; this research provides a more detailed framework for the specific problem of selecting the faults to seed into a software testing experiment. This research also addresses issues 4) and 5) by providing ways of controlling and comparing the faults used in experiments.

2.2.1. **Object of study**

Software testing experiments study the ability of code coverage criteria to find faults. Typically, the researcher will select a criterion, such as all-paths coverage, and compare its ability to find faults with a "null" criterion such as purely random testing.

2.2.2. **Hypothesis**

The null hypothesis is that the selected criterion finds about the same number of faults as the null criterion. The alternate hypothesis is that the selected criterion finds more faults than the null criterion.

2.2.3. **Inputs**

The experiment takes one or more programs as input. For each program, the experiment requires one or more faults; the intention is to see how well the test sets can find the faults. There are typically two sources of faults: faults from the program's revision history, and faults generated automatically by program mutation.

2.2.4. **Outputs**

The output of running a sample of test sets satisfying a given coverage criterion is a measure called Defect Detection Effectiveness. It is another name for the recall measure in Equation 4. Software testing experiments typically do not study precision. Test cases are generated from a correct version of the program, so it is assumed that the generated test cases do not make mistakes.

2.2.5. **Implementation**

There are two processes that the researcher must implement: 1) generation of test cases and 2) generation of faults.

Test cases

First, the researcher must generate one or more test sets satisfying the selected and null coverage criteria. To do this, the researcher must write or use an existing test case *generator*. The generator takes the grammar describing the input to the program (assumed to be correct) and iterates over the universe of possible accepted values. The researcher generates test cases and adds them to the test set until the desired level of coverage is achieved. The researcher then uses the same generator to generate another test set satisfying the null coverage criterion.

Faults

Second, the researcher must generate faults. If the program is under version control, the researcher will simply recompile the code under earlier versions to “generate” faults. In the absence of historical data, the researcher will use program mutation to generate faults. Program mutation makes syntactically small changes to the program, such as replacing the plus operator with the minus operator.

After the researcher generates the test sets and fault versions, the researcher runs each test set against each faulty version. For each test set, the researcher computes defect detection effectiveness to determine what percent of faulty versions were discovered by that test set. After

all test sets from both coverage criteria are executed, the researcher uses a hypothesis test to compare the mean defect detection effectiveness.

2.2.6. **Threats to validity**

In terms of the test sets used in the experiment, it could be the case that the generated test sets are not representative of real test sets. For instance, because the test sets are generated, the sets could be much larger than what would appear in industry; the issue is that no tester could realistically maintain an excessively large generated test set. Also, the test sets could have much higher coverage than what could be reasonably achieved in an industry setting; maybe it only took ten test cases to achieve 80% node coverage, but due to tricky conditions in the code, it took 1,000 generated test cases to achieve 100% node coverage. Research into *fault proneness* tries to prioritize testing of code that is the most likely to contain faults, thus alleviating the tester of the time-consuming problem of achieving 100% coverage.

In terms of the seeded faults, it could be the case that other programs will not experience the same faults in the same proportion as the proportions that were studied. If the researcher used the revision history to generate faults, then it could be the case that the faults are not representative of the faults found in other programs. If the researcher used program mutation, then the issue is that the faults may not be realistic. Research into *fault size* surveys and quantifies faults so that future researchers can select "representative" faults for their experiments.

2.3. Fault size

Several researchers have attempted to address the above threats to validity by specifically pursuing a way of quantifying fault size.

2.3.1. **Semantic fault model**

Offutt and Hayes [14] introduced two dimensions for measuring fault size: syntactic (number of tokens changed in the source code) and semantic (proportion of the inputs that catch the fault). They used this classification to explain several phenomena, such as the coupling effect and selective mutation. They performed an experiment to measure the semantic fault size of a tool's mutations. They mutated very simple programs that take a single floating point number from the Unix rand tool as input. They found on average that the mutants have about 31% semantic fault size. They also found that mutations had different semantic fault size depending on where the mutation was applied. While this is to be expected, this research posits that mutations have some

intrinsic detection difficulty and that all other variance can be explained away by accounting for factors external to the mutation.

2.3.2. **Mutation kill ratios**

Bertolino et al. [15] defined a set of mutation operators for access policy control to be applied to the eXtensible Access Control Markup Language (XACML) 2.0 language. They also developed a tool to generate mutations to XACML. The mutations include: errors in the specification of XACML functions, and errors in the ordering of the rules. The tool, XACml MUTation (XACMUT), generates mutants of XACML policies, executes XACML requests (test suites) on the original XACML policy and the mutated policy, and calculates the defect detection effectiveness. XACMUT also implements the access control policy mutants defined by Martin and Xie [16] as well as by Mouelhi, Fleurey, and Baudry [17].

Bertolino et al. did not specifically discuss the size or frequency of the mutation operators, but Table I in that paper indicated that three mutant operators from Martin and Xie are never applied and others are always killed. Other mutants occur frequently (over 60% of the time) and provide a 65% mean fault detection effectiveness. Mutation operators from Mouelhi, Fleury, and Baudry include one that is rarely applied (20% of the time). On average, the test suites applied to these mutation operators have low fault detection effectiveness (39%). Of their own mutation operators, Bertolino et al. report “the test suites reach the highest percentage of fault detection effectiveness (79%) showing that they can address most of the new conceived types of faults.” This thesis investigates how likely it is that similar mutation operators are killed.

2.3.3. **Relationship between mutants and real faults**

Offutt [18] hypothesized that "simple" mutants (comprised of one mutation operator) and "complex" mutants (combinations of mutant operators) are coupled. That is: if a test set can find simple mutants, then it can most likely find complex mutants. To test this hypothesis, Offutt applied mutations to TriTyp, Find, and Mid to generate mutant-complete test sets; these represent test sets that can find "simple" faults. He then applied two mutations at the same time to create complex faults. He found that the “simple” test set could still find 99% of complex faults [3]. The idea of fault size presented in this thesis provides a way to verify Offutt's hypothesis.

2.4. Spectrum-based Fault Localization

Spectrum-based fault localization (SBFL) is a relatively new method of software testing. Given an application and a set of unit test cases with some failures, one might ask: which lines of code are most likely causing test failures? SBFL utilizes various information retrieval similarity measures, such as cosine similarity, to compare execution traces and test failures. This comparison is believed to automatically infer the location of faults. This approach has the potential to create huge time savings for industry practitioners, who at present must analytically arrive at the location of faults.

In the envisioned use of spectrum-based fault localization in industry, practitioners prepare a set of test cases to run. The practitioners recompile their product source code in such a way that it records a log of when each statement of source code is executed. The practitioners run the test cases, one at a time, on the modified product. Each time a test case executes, the practitioners read the log and associate the executed statements with the test case. When all test cases have finished executing, the practitioner has two sets of data:

- A list of test results
- Logs of executed statements, organized by test case

The test results are encoded into a vector as follows: a "1" means that the test case failed, while a "0" means that the test case passed. For instance, the vector <1,0> means that there were two test cases in which the first test failed while the second test passed. The vector <0,0,0,1,0> means that there were five test cases and the fourth test case failed. Test cases usually run in a serial fashion, so it is easy to impose a total order on the test cases in this way. Research typically assumes determinism in the testing process, so for a given pairing of source code and test set, it is assumed there is a unique test results vector.

The practitioner converts the logs of executed statements into a set of similar vectors. Each vector represents the execution of a statement. Each dimension in the vector maps to the same test case in the test results vector. A "1" in the vector means that a statement was executed by the corresponding test case, while a "0" means that the statement was not executed by the corresponding test case. For instance, the vector <1,1> means that there were two test cases and both test cases executed the statement represented by the vector. Observe that all statement execution vectors will have the same dimensionality as the test results vector, so if there are five

test cases, all statement vectors will have length five as well. There may be arbitrarily many statement vectors; there will be one for each line of executable code in the product.

With these vectors available, the practitioner compares each statement execution vector with the test results vector and computes their similarity. At this point, the problem reduces to a typical information retrieval problem. For instance, it is common to compare vectors using cosine similarity [19]:

$$\cos(d, u) = \frac{d \cdot u}{|d||u|} \quad (1)$$

The practitioner compares each statement execution with the test results and assigns the statements the resulting cosine similarity. For instance, the cosine similarity between the test results <1,0> and the statement vector <1,1> is 0.707. The practitioner sorts the similarities and chooses the statement with the highest similarity. This statement should, with no guarantee, contain the most likely cause of the observed test failures. If the practitioner sees this statement and does not think that it caused the fault, he or she assesses the other statements in descending order of their similarity score until arriving at the true problem statement.

Abreu et al. [19] examined the recall and precision of this technique. They reduced the problem to two inputs: a) the similarity measure and b) the number of failing test cases. Given this context, they evaluated two aspects of the problem: the similarity coefficient used to compare the vectors and the number of failing test cases. They used several tools to compare the similarity measures, including Tarantula (cosine), Jaccard, and Ochiai. They found that Ochiai led to minor improvements in the recall and precision of the results, but found that more test failures led to substantial improvements regardless of the similarity measure [19].

Experiments in this area take the above form. As experiments show, this technology is not perfect; there are some faults that are not correctly located. Suppose a fault f is injected into the code. Given the code's test set, two questions arise in SBFL:

1. How many test cases execute f? (denote as T)
2. How many test cases in T identify f? (denote as T_f)

SBFL expects that T and T_f will be roughly equivalent. For "easy to find" faults, this expectation is valid. For "hard to find" faults, this expectation will cause the similarity scores to appear artificially low, potentially leading to false negatives. Unfortunately, in all existing validation of SBFL, there is no regard for the nature of the fault f being localized; these techniques are not

fault-based. It is not clear whether this technique generalizes to all types of faults. An understanding of the nature of the fault, in particular a method of quantifying its *fault size*, would be very meaningful to eliminating this threat to validity. If researchers could account for how easy their faults are to find, they could assess the true effectiveness of the various information retrieval algorithms being proposed. The thesis research decomposes this notion of fault size into several probabilistic events that occur when a fault is discovered.

2.5. Fault Proneness

Ideally, practitioners would correctly prove the correctness of all their code. Failing that, they would exhaustively test all of their code. For large systems, neither approach is feasible. Thus academics have introduced this notion of *fault proneness* as a way of prioritizing quality assurance activities. The common vision is to start testing activities with the most fault prone code, then continue testing less fault prone components until testing reaches a target fault proneness. The open question is: what criteria should be used to classify code as fault prone?

2.5.1. **Program vocabulary**

The complexity of a program's vocabulary has been a strongly suspected source of fault proneness. This section examines recurring ideas in fault proneness literature with respect to program vocabulary.

The Halstead code complexity metrics are well-known vocabulary-based metrics. Whereas code complexity generally connotes algorithmic complexity, Halstead thought of code complexity in terms of the breadth of the program's variables and operators. Two fundamental Halstead metrics are:

- *Vocabulary*: the number of distinct operators and operands.
- *Length*: the number of operators and operands.

Using these definitions, Halstead created a number of derivative metrics, such as “difficulty” and “effort,” which estimate the time required to develop software [20]. A number of studies use these metrics to link code complexity to fault proneness [21]–[23].

Hata et al. claimed to be the first to examine fault proneness on the per-line level. Their approach borrowed from the vocabulary based methods of spam filtering. They trained a machine learner on the text of fault-prone lines, and then looked for files containing lines with similar vocabulary.

They back tested their work by training the machine learner on Eclipse changesets. They used the changesets to predict which modules in Eclipse would contain faults. To assess their results, they measured recall and precision on the per-module level. They surveyed other research to compare their results [24].

On the module level, Hata et al. stated high recall and precision. Unfortunately, the results did not fit the granularity of the estimator. The estimator was trained to identify specific problem lines, but Hata et al. did not state how well their classifier identified the actual faulty lines [24]. Thus it is not clear how well Hata et al. identified specific fault-prone lines.

This thesis research is orthogonal to vocabulary metrics. This research relies on the code's structure, which uses no information at all from the code's vocabulary.

2.5.2. **Cyclomatic complexity**

McCabe introduced a seminal fault proneness measure called cyclomatic complexity. Given a control flow graph with one entry and one exit node that are not strongly connected to each other, McCabe defined cyclomatic complexity as:

$$M = E - N + 2 \tag{2}$$

where E is the number of edges and N is the number of nodes in the graph. McCabe claimed this number represents the number of "linearly independent paths." To simplify testing, McCabe proved that code transformations can reduce this complexity, but did not give guidance as to which transformation is optimal at a given point. McCabe claimed that a function with a cyclomatic complexity of 10 or more needs to be tested [25], [26].

The thesis research closely relates to cyclomatic complexity. This research introduces a new measure, testability, that similarly measures the structural complexity of the code. This research advances cyclomatic complexity in two ways. First, this research assigns complexity values on a per-line basis. McCabe only assigns complexity values on a per-function basis, so it is not as useful for guiding developers to specific problem areas. Second, this research uses a more informative source of information than cyclomatic complexity. Cyclomatic complexity counts the number of linearly independent paths, but this research uses actual test paths from an open-ended source, such as actual test data or a test case generator. This research demonstrates that these two factors help better predict the locations of bugs than cyclomatic complexity.

2.5.3. **Software testability**

Testers sometimes encounter difficulty testing code because the code's state is unobservable, for instance because the state is hidden in local variables that disappear after the program is finished executing. Developers write unobservable code for a variety of reasons, in fact the information hiding paradigm encourages it. While testers could use debuggers to step through code to verify the computations are accurate, debugging is indiscriminate and thus excessively expensive; there needs to be a way to predict which modules need more testing. This prediction is generally called *software testability* and provides another form of fault proneness prediction.

Three research groups separately investigated software testability. This section discusses the two approaches and how they relate to the thesisresearch.

Pisces

Voas and Miller examined the notion of software testability. They describe information loss issues in software development that make code difficult to test. For instance, computations performed on local variables may not be visible to external testing code. To identify code that is difficult to test, Voas and Miller introduce a dynamic analysis technique called sensitivity analysis, implemented by a tool called Pisces. Sensitivity analysis is a synthesis of three types of dynamic analysis: “execution analysis” (out of X random test runs, in how many of those runs was a given block of code covered?), “infection analysis” (aka weak mutation testing; does the mutant cause the internal state to change at the point of the mutation?), and “propagation analysis” (aka strong mutation testing; does the mutant cause the output to change?). They call this approach P.I.E. for propagation, infection, and execution, respectively [27], [28]. Voas independently evaluated his approach by correlating execution probability to random execution. He found a very strong correlation between his predicted execution probability and code covered by random execution [27].

This thesis uses a novel technique for computing the same sort of execution probability as that of Voas and Miller. A key difference is that their approach is dynamic, requiring many executions of the code under test. The issue is that the existing set of test cases may not be sufficiently large enough to produce a testability estimate within a reasonable margin of error (they list the margin of error and required sample size at several thresholds). This thesis uses static analysis and does not require any test cases.

Measuring propagation

Freedman addressed the problem of the lack of propagation of variables in testing and created *domain testability* to measure the lack of propagation. Freedman formalized two desired qualities of domain-testable programs: observability (the program under test is purely a function of its parameters; it uses no global state) and controllability (the program can output all values in its defined range). Freedman measured domain testability by creating a domain-testable version of the program and then counting how many parameters he had to add; the more added parameters, the worse the domain testability. Freedman's students evaluated the measure by coding and testing programs according to specifications written with and without domain testability in mind. Freedman found that domain-testable specifications help speed up development [29]. This research is different from Freedman's because it focuses on simplifying the code's structural properties instead of restating the code's domain and range.

Santelices and Harrold [30] revisited the problem of estimating evaluation and propagation of an error in the context of program slicing. They observe that propagation is linked to data flow dependencies. Often static data flow analysis overestimates data flow dependencies, while dynamic data flow analysis underestimates the data flow dependencies. To reduce the number of dependencies found in static analysis, they introduced a probabilistic model of def-use propagation. They use this model to rank and select the top results. This thesis reuses their def-use propagation model to help explain fault size.

Automatic path construction

In the absence of an actual test set, a static estimate of code's likelihood to be executed can determine its testability. Li et al. examined algorithms for reducing coverage requirements into a smaller set of actual test paths. When graph-based coverage criteria produce many requirements, sometimes there is overlap in the requirements, causing the criteria to overestimate the amount of work required to test the program. Li et al. cast the problem of minimizing coverage requirements into a prefix graph. Li et al. detailed algorithms to extend the criteria into full test paths. They provided a tool, `Graph-Coverage`, for use in our research [31]. This thesis uses this tool to develop a novel method of determining code testability.

Filieri et al. [32] examined the use of symbolic execution to estimate a program's failure probability. Given a probability distribution representing the input domain, their technique can determine the likelihood that a program will produce a runtime error. They implemented their

symbolic execution as a plugin in NASA's Java Path Finder (JPF) [33] for finite domains (integer, Boolean, etc.). While the Filieri et al. tool appears to measure fault size, it performs poorly in the context of measuring the fault size of program mutations. The Filieri et al. tool returns different fault size measures for different programs, even if the same mutation operator, such as replacing less than with less-than-or-equals, is applied. Thus their tool is not useful for verifying the thesis of this research; our research instead presents a more detailed model that is readily applicable to program mutation.

2.5.4. **Evaluation techniques**

Several researchers have combined vocabulary and structural fault proneness predictors using linear regression to form even better fault proneness predictors (to date, no one has used the testability measures to build such a model). Researchers typically evaluate the regression model's quality using three measures: Pearson's correlation coefficient [34], recall, and precision [35].

To measure recall and precision, researchers build a confusion matrix listing true positives, false positives, true negatives, and false negatives. To delimit “positives” and “negatives,” they typically select an arbitrary cutoff for the predictor variables, such as 0.5 on a [0,1] fault proneness scale (in fact the goal of the experiment is usually to identify the optimal cutoff). Anything higher than the cutoff is considered a positive (prediction: it will have a bug) while anything lower than the cutoff is considered a negative (prediction: it will not have a bug). To determine whether these predictions are correct, researchers examine the code's revision history. Code that contained a bug in the past is considered to be faulty. All other code is considered not faulty. To measure Pearson's correlation coefficient, researchers pair the bug history with the fault proneness predictors under study and perform a least-squared fit; the technique is well-known.

Lanubile et al. combined a number of simple fault proneness metrics into a linear model. These metrics include: McCabe's cyclomatic complexity [25], Halstead metrics [20], and Henry and Kafura's fan in/fan out [36]. They trained the linear model on the student projects from a software engineering course. They used the model to build a confusion matrix and computed chi-squared to determine if the predictions were better than random. They used a chi-squared test to compare the confusion matrix to random classification. They found that their model did not perform significantly better than random classification [22].

Bellini et al. built two linear models using two large collections of fault proneness metrics, CPP-Analyzer and PAMPA. Using both tools, they built the two linear models for two programs (one

model per program). Using principal component analysis, they reduced over one hundred predictors down to only eleven principal predictors for one program and five for the other. Sadly, due to a lack of space, the authors could not state what these metrics actually were. The Pearson correlation coefficients were moderate at 0.5 for the first model and 0.6 for the second. The authors did not cross validate these models. When the authors fitted both data sets to the factors in common, the correlation slipped to only 0.4 [21].

Ostrand et al. tried to predict *how many* faults each file in a program will have. Their linear regression modeled many factors, including: the log of the number of lines of code, the log of the number of faults, the file type, and the product release number. They found that their model poorly predicted the actual number of faults (to such an extent that they would not state more than this). To evaluate their model's recall, the authors set a threshold to include only the top 20% of files with the highest predictions. The authors found that this top 20% contained on average 83% of all faults [23]. This presentation of the results is not confidence inspiring. First, the authors did not report precision. It could be that their predictions have many false positives. Second, the authors said that the top 20% of files according to their model were also the longest files. The issue is that they forgot to prove that the top 20% of files did not also contain 83% of all lines of code. All lines of code being equally likely to be faulty, one would expect random selection to assign the most faults to the largest files.

These derived models show that existing fault proneness metrics still fundamentally lack the ability to predict faults. This result motivates the need for better fault proneness metrics.

2.6. Statistical Conclusion Validity

In a typical software reliability paper, researchers introduce a new technique for finding errors in code, a.k.a. "bugs" or "faults." To show that their technique is significantly better than the state of the art, they evaluate their technique as well as the state of the art technique on one or more programs that are known to contain faults. They then try to establish that their technique finds "significantly" more faults than the state of the art, using statistics to validate this claim. Unfortunately, software reliability engineers are generally not statisticians and do not have statisticians readily available. Statistics software such as R provides several viable hypothesis tests for performing statistics, but for a researcher without a statistics background, picking a test at whim is fraught with potential for error. Many tests make strong assumptions about the shape

of the data; without testing those assumptions, the results from hypothesis tests are meaningless. While tests exist that make few assumptions, these tests are less powerful and are less likely to indicate that research is statistically significant when researchers know it is significant for other reasons.

2.6.1. **Definitions**

To evaluate a hypothesis, researchers in software testing typically look at one or more *summary statistics*. The most common statistic is the *mean*, or average. Researchers strive to show that one approach can find more faults on average than another approach. Note that in this section, the term "test" refers to a statistical test and not dynamic execution of software as part of software testing.

According to Cohen, *effect size* is a key measure in computing an experiment's *power*, which is its ability to correctly reject the null hypothesis [37]. Often, researchers will run small, initial experiments to see if their ideas hold any merit. While they may find a positive result, their result might not be statistically significant. *Post-hoc power analysis* is a procedure used at the end of such a pilot study to determine the sample size required to achieve a statistically significant result. The analysis assumes that the effect size the researcher observed will remain constant in the larger experiment.

This thesis refers to several hypothesis tests for comparing means. Well-known hypothesis tests include the *t-tests* [38] and the *Wilcoxon* tests [39]. Less well known is the recent *Brunner-Munzel* test [40]. The t-tests are considered *parametric tests* because they assume parameterized families of distributions (normal distribution family or others where a particular distribution can typically be described using one or two parameters). The Wilcoxon and Brunner-Munzel tests are considered *nonparametric* because they do not assume that the data originates from any particular distribution family.

In particular, the t-tests assume *normality*: the assumption that the data fits a normal distribution, also known as a "Gaussian distribution" or "bell-shaped curve." The normal distribution is defined to be *symmetric*, meaning that the mean and median are equal. Symmetry does not always hold; distributions like the beta distribution can be *asymmetric*, which indicates a significant departure from normality. Distributions can also have varying *peakedness* ranging from the completely flat uniform distribution to the especially sharp t-distribution. Pearson [41] formally defined these aspects of non-normality in terms of *skewness* (asymmetry) and *kurtosis*

(peakedness). Many methods of varying effectiveness exist for verifying the normality of data, but we use the *Shapiro-Wilk* test [42] because it formally reasons about the likelihood that a given sample's skewness and kurtosis are a significant departure from normality.

Some tests assume that the data takes the form of *two independent samples*. One might imagine a software testing experiment where testers are divided into two groups (control and treatment group) and testers in the treatment group are given a new method of detecting faults. We might then formulate a hypothesis that testers in the treatment group find more faults than testers in the control group; this would be an example of an experiment with two independent samples. Other software testing experiments take the form of *paired sample* experiments. In these experiments, each defect detection algorithm under study is applied to every program. The defects found are related (paired) on the program being tested. The experiment designer establishes whether their experiment is paired or two-sample based on the nature of the procedure; it cannot be inferred from the samples. The choice of two independent samples *vs.* paired samples determines which hypothesis tests can be applied.

Some independent samples tests, like the classical Student's t-test, also assume that the samples have *equal variance*. Several formal hypothesis tests exist for disproving this assumption, such as the classical F-test and the more robust Brown-Forsythe test [43]. However, the F-test itself makes strong assumptions about normality, so in cases where normality is not clearly inferred from the sample data, Brown-Forsythe can be more appropriate. When the data does not have equal variance, Welch's t-test is appropriate because it computes a more robust estimate of the variance.

The Brunner-Munzel implementation in R [44] uniquely assumes that there is *overlap* between the samples. If one were to plot two independent samples on a shared number line, the number line would indicate overlap if at least one value from one sample fell within the min and max values of the other sample. When no overlap is present, our experience is that the Brunner-Munzel test is undefined. Overlap is likely a new assumption to researchers, but as the description indicates, overlap is straightforward to test.

2.6.2. **Statistics in software engineering**

Statistics are frequently used in the design of software engineering experiments. This section discusses efforts to survey their use.

Arcuri and Briand surveyed the presence of statistical analysis practices in the study of the performance of randomized solutions to computationally hard problems. In the 16 papers studied, they found that only five papers used statistical analysis. Of those, three used linear regression, one used the Mann-Whitney U test, and one used both the U test and the t-test. Through an analysis of hypothetical scenarios comparing the t-test with the U test, they argue for the use of the U test in most scenarios. They provide a "practical guide" to using statistics in software engineering based on these findings [4]. This thesis refutes the conclusions of this guide.

Kampenes et al. [45] systematically reviewed the statistical significance of 103 controlled software engineering experiments dating between 1993-2002. Their goal was to examine the proportion of journal papers that report Cohen's effect size. As a secondary measure, they collected the number of statistically significant results as reported by the authors. The 103 papers contained a total of 429 hypothesis tests. Of those, only 212 tests (49%) indicated statistically significant results. This proportion serves as a baseline for a meta-analysis of ICST'13 in Chapter 7.

Chapter 3. Summary of Techniques

This chapter provides a summary of the new techniques defined by this research, including the model of fault size and the MeansTest algorithm. This chapter also summarizes the research questions that this thesis research answers.

3.1. Fault size

The background concepts and necessity of fault size were described in section 2.3. This section provides a description of the fault size model presented in this research. The complete formalisms are developed in Chapter 8 and Chapter 9.

As previously discussed in Chapter 2, Voas already established that three events are necessary for a fault to be found [27]:

1) Execution: the code containing the fault must be executed.
2) Infection: the code must infect the local data state.
3) Propagation: the infected data state must propagate.

This thesis research formally models these events and their interaction in the language of set theory. In essence, a fault can only be found when the intersection of the above events occurs. Figure 3.1 abstractly shows the interaction of these events in the discovery of a fault.

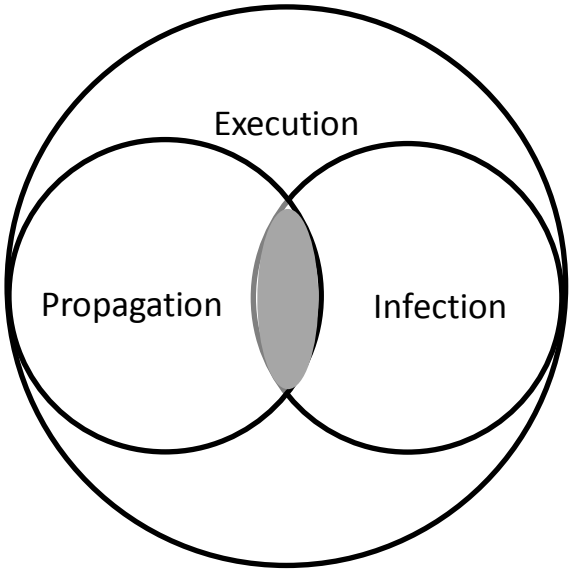

Figure 3.1. Model of fault size

As the diagram shows, faulty code can be executed, but not infect the data state. Similarly, faulty code can be executed and infect the data state, but its result may not propagate to some point readily verifiable. As well, faults cannot infect the data state nor propagate if the code is not executed. Spectrum-based fault localization implicitly assumes that executing faulty code is sufficient to achieve the required intersection of the three events, but as the diagram shows, this is not necessarily the case; it depends on the true size of the infection and propagation spaces.

In the context of software testing, the execution probability is the likelihood that code is tested; this thesis research calls this probability *testability* and formally investigates possible definitions in Chapter 7. In essence, there are two main ways of determining testability. One can *dynamically* profile the proportion of time that an existing test set executes the code. This assumes that the test set executes the code in a way that reflects the code's real-life usage. In the absence of this type of test set (which is often the case in research) one must *statically* estimate how likely the code will be executed. This thesis describes a novel method of statically estimating testability. The method analyzes the possible paths through the code's logic. To determine these paths, the method uses the static path analysis by Li et al. [31]. When the majority of paths execute a particular line of code, this thesis attributes *high testability* to this code. When the minority of paths execute a particular line of code, this thesis attributes *low testability* to this code [26].

This thesis research describes a novel procedure to analytically derive the size of the infection space in the diagram; this quantity is referred to as *intrinsic size*. If a fault has a particularly small infection space, it biases the experiment against finding the fault; this thesis calls this type of infection *intrinsically small*. If a fault has a particularly large infection space, it biases the experiment in favor of finding the fault; this thesis calls this type of infection *intrinsically large*. Worked examples showing the computations involved are given in Chapter 9.

The propagation aspect is well-studied in this field and is presented in Chapter 8. In essence, propagation is guaranteed if there is a path of definitions and uses from the infected line of code to some use in a test case. Otherwise, propagation is not guaranteed; it can only be certain if the infection alters the path taken through the program [30].

3.2. MeansTest

The MeansTest algorithm automates certain steps of the statistical analysis process in software testing research. In a typical software testing experiment, one collects two data samples (number of faults found by old and new testing methods), then tests several conditions concerning the shape of the data. The conditions one examines are:

- **Normality**: whether the number of faults found is bell-shaped;
- **Equal variance**: whether the number of faults found has equal scatter around the mean; and
- **Overlap**: whether there is intersection in the two samples.

MeansTest is unique in the way it makes determinations about these factors. Classically, a statistician would visually examine the shape of the data to determine which conditions hold. An automated solution would be preferable, but statisticians are aware of the uncertainty in this endeavor, so formulate such solutions in terms of specialized statistical hypothesis tests. The language used is familiar to researchers: the null hypothesis is that the condition holds and it is disproved when the likelihood that the null hypothesis holds falls below a critical value such as 5%. MeansTest combines select tests into a workflow for determining which conditions hold; the validity of these selections is examined in Chapter 10. These tests are:

- Normality: **Shapiro-Wilk** [42];
- Equal variance: **Brown-Forsythe** [43]; and
- Overlap: an original test.

Using this information, MeansTest also contributes a workflow for determining the appropriate hypothesis test:

- Normality and equal variance: **Student's t-test** [38];
- Normality holds but not equal variance: **Welch's t-test** [46];
- No normality but overlap: **Brunner-Munzel** [40]; and
- No factors hold: **Wilcoxon** procedures [39].

As mentioned in Chapter 1, this type of workflow is rarely used in the software testing field. Instead, researchers tend to choose the last family of tests in the workflow, the Wilcoxon tests,

because they require that none of the above conditions hold. Arcuri and Briand asserted that the Wilcoxon tests are more readily applicable than the t-test to the data that software testing researchers examine, [4] implying that such a workflow is not necessary. That being said, in situations where the conditions **do** hold, the Wilcoxon procedures can be shown to be asymptotically weaker than the t-test, having only 96% asymptotic relative efficiency in normal distributions [47]. Furthermore, Brunner and Munzel [40] introduced a new test that this thesis research shows is superior to the Wilcoxon approach in Chapter 10.

3.3. Research questions

Table 3.1 summarizes the research questions and the respective chapters in which they are investigated below.

Table 3.1. Research questions

	Research questions		Chapter
Fault size	RQ1	How does the testability of fault-prone (FP) code compare with the testability of non-fault-prone (NFP) code?	Chapter 7.
	RQ2	How well do static fault proneness metrics predict the fault-proneness of code?	Chapter 7.
	RQ3	What are the fault sizes of typical mutations?	Chapter 8.
Statistical analysis	RQ4	How should statistical analysis be performed and presented in software testing published work?	Chapter 9.
	RQ5	How accurate is the MeansTest hypothesis test compared to classical hypothesis tests?	Chapter 10.
	RQ6	How prevalent is Wilcoxon statistical analysis in software testing research?	Chapter 10.
	RQ7	How often are results in software testing research significant?	Chapter 10.

As Table 3.1 shows, this thesis research started by investigating the individual aspects of the fault size model. As this research was conducted, it soon became evident that statistical analysis problems existed in software testing research at large that needed to be addressed first. Thus, this research was supplemented by research questions corresponding to observed issues.

Chapter 4. Research synthesis

This chapter summarizes the findings of each of the dissertation publications in a consistent format. In each section, the research questions, main findings, potential applications, and threats to validity are listed. More information can be found in each paper's respective chapter.

4.1. The Effect of Testability on Fault Proneness

RQ1: How does the testability of fault-prone (FP) code compare with the testability of non-fault-prone (NFP) code?

RQ2: How well do static fault proneness metrics predict the fault-proneness of code?

Main findings: FP code in the Apache HTTP Server had significantly higher testability than NFP code. The result was surprising because it was assumed that FP code would have lower testability. The interpretation is that highly testable code is easiest to test, therefore *known* bugs appear most often in easily testable code. It could still be the case that there are more bugs in untestable code that have not been found. However, this presents an ontological problem because it is not possible to construct an experiment from known bugs to represent the locations of unknown bugs.

Static fault proneness metrics, including the well-known McCabe cyclomatic complexity, were shown to have low precision, making many predictions about the fault-proneness of code that did not actually contain faults. The testability measure introduced by this research had superior recall to McCabe, but similarly low precision. The finding regarding McCabe cyclomatic complexity is significant because it refutes work by Ostrand and Weyuker [23]. One might alternatively interpret the Ostrand and Weyuker results as finding that no particular lines of code are faultier than others.

Potential applications and results: While the testability measure introduced in the work was not found to be particularly good at predicting the locations of bugs, it was able to accurately predict the likelihood that code was executed. This property plays well into the execution probability needed to estimate fault size.

Threats to validity: The main threat was to conclusion validity. The study did not examine every fault in Apache, so the precision values were all artificially low. This threat is mitigated because the recall values were not affected, so the main conclusions still apply.

4.2. A Framework for Assessing the Validity of Mutation-based Experiments

RQ3: What are the fault sizes of typical mutations?

Main findings: MuJava generated 341 mutants for the program TriTyp from seven mutation operators. Roughly 71% of the faults, or five out of the seven operators, had an intrinsic fault size of 100%, indicating that they were theoretically trivial to find through testing. The hardest category to find was the Relational Operator Replacement operator, which replaced relational operators with one another. Its intrinsic size varied depending on the specific replacement, but the average size was 50%.

A problem framework was presented for researchers to determine the fault size of real faults. The framework is left as future work for other researchers to apply.

Potential applications and results: The distribution of mutant fault sizes could be used as a baseline for comparing the size of real-world faults to mutant faults. If there is statistically significant disparity between the distributions, then it would give rise to the possibility that program mutation is not a sufficient way to test new theory. The model of fault size could also explain why spectrum-based fault localization techniques have trouble finding some faults but not others.

Threats to validity: The main threat was to external validity. The mutants analyzed were all from the Trityp program, which is easily analyzed, but is very small, written in one language, and is restricted to 32 bit integer operations. Each program has a different proportion of the different types of mathematical operators, so it is possible that the distribution of faults does not generalize to all programs. It would take many mutants from many sample programs to determine a more representative sample.

4.3. Statistical Analysis for Traceability Experiments

RQ4: How should statistical analysis be performed and presented in published work?

Main findings: The paper introduces the MeansTest TraceLab component for performing statistical analysis. The paper also suggests specific writing styles for expressing the results of the MeansTest analysis in appropriate statistical terms.

Potential applications and results: The MeansTest algorithm was written as an example of a complicated TraceLab composite component; feedback from other TraceLab component writers

indicates that researchers will do well to follow its implementation as an example for future components. The algorithm and writing style will revolutionize the way experiment results are presented.

Threats to validity: As of the writing of this paper, the MeansTest algorithm had yet to be empirically validated. The MeansTest algorithm is not a panacea; it does not automate all aspects of statistical testing and is not a substitute for an expert statistician. While the MeansTest algorithm contains the most robust statistical assumption tests known, they are not 100% accurate, particularly at small sample sizes of less than twenty.

4.4. Validation of Software Testing Experiments

RQ5: How accurate is the MeansTest hypothesis test compared to classical hypothesis tests?

RQ6: How prevalent is statistical analysis in software testing research?

RQ7: How accurate is statistical analysis in software testing research?

Main findings: The study compared the accuracy of the MeansTest component with four other hypothesis tests: Mann-Whitney, Student's t-test, Welch's t-test, and Brunner-Munzel. The study applied them to predetermined distributions, then ranked their ability to determine statistically significant differences. The study showed that the four classical hypothesis tests had at least one distribution for which they each performed the worst at finding statistically significant differences. The MeansTest algorithm was always ranked at least third, demonstrating superior worst-case performance.

Only 33% of the empirical papers at the Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST'13) featured any kind of statistical analysis. Of those papers, only 50% of papers accurately stated the significance of their results. One paper in particular was shown to have under-stated the significance of its results due to incomplete testing of statistical assumptions.

Potential applications and results: The results of the MeansTest analysis are very positive because they show that researchers can perform statistics more reliably with MeansTest than they could have by relying solely on the Mann-Whitney test, which was the state of practice for many years. Additionally, the paper featured a revised implementation of the MeansTest algorithm that performed post-hoc power analysis. The post-hoc analysis helps researchers select an appropriate

sample size in future work. Finally, the MeansTest used in this experiment was built in R. R is a common statistics language and so this implementation is more widely accessible than the previous TraceLab implementation.

Threats to validity: There is a threat to conclusion validity regarding RQ5 in that the study did not examine every possible distribution. The study minimized selection bias by using a predetermined set of distributions from another similar study by Razali and Wah [48]. There is also a mono-measure threat to validity in that we only assessed RQ7 using MeansTest.

Chapter 5. Future Work

Work will continue into the evaluation of fault size and the MeansTest algorithm. The next sections discuss the possibilities in each subject.

5.1. Fault size

Now that fault size has been formally modeled, the opportunity arises to apply the model to existing research. First, future work will involve reconstructing other researchers' experiments and determining the fault size distribution of those experiments. Some experiments will feature mutation while others will feature naturally occurring faults; future work will study the age-old question of the similarity or dissimilarity of these types of faults once and for all.

With this problem solved, work will turn toward assessing the size of faults that occur in practice. Naturally occurring faults again provide perspective on the size of faults, so rather than randomly sampling them as in experiments, a systematic survey of faults can occur. When this survey is complete, differences in fault size profiles between this universe of faults and experiments should indicate whether there has been any significant sampling bias in experiments.

5.2. Statistical analysis

The design of experiments on the statistical side will continue to be of interest. MeansTest automates the comparison of means, but other types of statistical analysis, such as analysis of variance and power analysis, exist. Analysis of variance has several parametric and nonparametric options that each has their own assumptions that need to be tested. It would be worthwhile to try to develop a similar automated algorithm to assist with this process. Power analysis is trickier to automate in the same way because there is no such thing as non-parametric power analysis, so considerable theoretical advances in statistics would need to be made first.

Chapter 6. Conclusions and Main Contributions

While there are many ways that software engineers propose to improve software quality, in practice software testing is the approach that is most easily understood and thus used in practice. This ease of use does not erase the substantial threats to validity to the process of software testing. These threats manifest in very real and costly ways, as was the case with the healthcare.gov failure to launch. While practitioners tout the latest and greatest software testing techniques, appropriate scientific rigor in the underlying theory is essential to making a real difference in the state of practice in industry. Failure to uphold this rigor can result in testing services and products that only work in theory, not in practice.

The research presented in this dissertation contributes to the understanding of the construction of software testing experiments. Contributions consist of a formal model of fault size, an algorithm for automating some parts of statistical analysis, an empirical study of static analysis metrics, and an empirical study of hypothesis tests and their application in software testing research.

The **formal model of fault size** finally formulates existing notions of code testability and the semantic fault model into a succinct definition. The definition breaks down a highly abstract problem into several areas, some of which have already been well-studied. The notion of intrinsic fault size is a particularly novel contribution that has the potential to separate experiment noise from true deficiencies present in new testing theory such as spectrum-based fault localization.

The **MeansTest algorithm** for automating some parts of statistical analysis vastly improves the state of the art in software engineering. Researchers have made dangerous generalizations regarding the applicability of certain results and hypothesis tests within our field. While any result from MeansTest theoretically suffers the same threats to validity, its worst-case performance has been demonstrated to be superior to the current practice. More importantly, its simple and transparent TraceLab implementation will hopefully invite curiosity from other researchers, encouraging them to think more critically about how they perform their statistical analysis.

The **empirical study of static analysis metrics** shows that static analysis methods that measure fault proneness, such as McCabe cyclomatic complexity, are far from the panacea that other researchers have claimed. It shows deep flaws in the experiment methodology used in past experiments. While the failure of the novel testability measure is a negative result, the testability

measure has been shown to have other approximations based on spectrum-based fault localization that better approximates the testability of the code.

The **empirical study of hypothesis tests** shows that no hypothesis test is perfect in every situation. That being said, the study suggests that MeansTest is never the worst option. For researchers who insist on choosing the same hypothesis test for every experiment, MeansTest will mitigate the most damage. The study of the application of hypothesis testing in software testing research confirms that many researchers do not know how to validate their work.

In conclusion, the empirical studies show considerable flaws in current software testing research. The formal model of fault size and the MeansTest algorithm have the potential to raise the quality of scientific rigor applied in this field. Through increased awareness of the effects of fault size on experiment outcomes, researchers can better calibrate their selection of faults to more thoroughly assess new testing theory. Similarly, the MeansTest algorithm will bring awareness of new options for testing statistical assumptions. These advancements will advance the state of software testing research, improving the results of its practice in ways that can be materially felt by all.

Chapter 7. The Effect of Testability on Fault Proneness

7.1. Summary

Numerous studies have identified measures that relate to the fault-proneness of software components. An issue that practitioners face in implementing these measures is that the measures tend to provide predictions at a very high level, for instance the per-module level, so it is difficult to provide specific recommendations based on those predictions. We examine a more specific measure, called software testability, based on work in test case generation. We discuss how it could be used to make more specific code improvement recommendations at the line-of-code level. In our experiment, we compare the testability of fault prone lines with unchanged lines. We apply the experiment to Apache HTTP Server and find that developers more readily identify faults in highly testable code. We then compare testability as a fault proneness predictor to McCabe's cyclomatic complexity and find testability has higher recall.

7.2. Introduction

The importance of fault proneness metrics is usually explained with succinct reasons, such as "methodologies and techniques for predicting the testing effort, monitoring process costs, and measuring results can help in increasing efficacy of software testing" [49]. These reasons make assumptions about the importance of software testing that outside observers may not share. We hear anecdotal stories from local software development firms where clients state during negotiations that they refuse to pay for time spent on testing, including the types of tests that test-driven development involves (such as unit tests).

During negotiations, their clients cite two issues with paying the developers to test. First, they see no reason why highly paid and qualified developers should make errors. Second, they believe that testing performed by the developers presents a conflict of interest. A series of Dilbert comics parodies this concern, where the developers are told they will earn $10 for every bug fixed [50]. From the client's perspective, the developers are no different from Dilbert. These two issues give clients pause and thus our local development companies simply see no reason to train developers to test, much less hire designated quality assurance staff.

Inspired by this problem, we issue a challenge to the fault proneness community to use their huge collection of metrics to propose actionable development plans to improve code quality. In the same vein as formal specification, rather than *testing* more, we propose *developing* more to reduce the testing burden of proof. To this end, we describe a novel fault proneness metric based on previous work in test case generation. Our contributions in this paper include: our position on

using fault proneness metrics to improve code quality, our new testability metric, our tools, and our experiment.

In Section III, we describe our metric, *testability*, and illustrate our vision of how a developer could use certain fault proneness metrics to systematically reduce fault proneness. In Section IV, we describe a case study performed on the Apache HTTP Server and discuss our surprising results.

7.3. Testability

We define testability with an intuitive static approximation to the Voas execution probability. For each function in the software, we generate the function's control flow graph. We feed the graph to the `Graph-Coverage` tool to compute the minimum set of paths required to achieve node coverage. We overlay all of the paths onto the graph. We define the testability of each node in the graph as the proportion of paths passing through that node. A testability of 1 means the node has perfect testability. A testability of 0 means the node is unreachable.

7.3.1. **Formal definition**

Let $\text{GraphCoverage}(G, \text{NODE})$ be the minimum set of test paths satisfying node coverage for graph *G*. Let $\text{GraphCoverage}(G, \text{NODE})_b$ be the subset of $\text{GraphCoverage}(G, \text{NODE})$ test paths containing node *b*. We define the testability of node *b* as:

$$t(b) = \frac{\text{GraphCoverage}(G,\text{NODE})_b}{\text{GraphCoverage}(G,\text{NODE})}. \quad (2)$$

`Graph-Coverage` refers to the Li et al. prefix graph algorithm mentioned in the related work. It can be substituted for an equivalent program; for instance, NASA's Path Finder would suffice. Similarly node coverage could be substituted with a stronger criterion (node coverage happens to scale well from a computational standpoint). We also envision practitioners extracting test paths from their operational profiles and/or test sets in place of `Graph-Coverage`. In short, there are many possibilities. We challenge the community to examine the effectiveness of these other techniques in place of our static approximation.

7.3.2. **Implementation**

We have developed tools to compute the above testability definition for arbitrary C and Java code. We developed the tools with maximum reuse of common Linux tools in mind. We hope that practitioners will benefit from our insights into developing tools of their own.

The tools build a database mapping the source code line numbers to the control flow graph. To build the database, the tools parse the compiler's internal control flow graphs. From this database, the tools computes the testability scores by invoking `Graph-Coverage`, parsing that format, then computing (2). To get human-readable results, the tools perform a join operation to map the `Graph-Coverage` output back to source line numbers. They also generate a GraphViz [51] map.

The tools defer the parsing of C code to the `gcc` compiler [52]. The compiler can dump the control flow graph at many stages of compilation. We decided to use the Single Static Assignment (SSA) representation [53]. The SSA dump is useful because it explicitly lists the *basic blocks* (atomic groups of lines), the edges, and the mapping from block to source lines. For very small projects, practitioners can generate the SSA dump for their own purposes using:

```
gcc -g -fdump-tree-ssa-lineno-blocks
```

Several open source projects use `configure` scripts that configure the compiler for all files. The following command (on one line) configures the compiler to create a SSA file for every file that it compiles:

```
./configure CFLAGS="-g
-fdump-tree-ssa-lineno-blocks"
```

With the SSA file available for each compiled C file, our tools simply use `awk` [54] to parse the SSA line-by-line. From the resulting database we can easily generate the graphs in the adjacency list format that `Graph-Coverage` expects.

The `gcc man` page seems to favor `-fdump-tree-cfg` and `-fdump-tree-vcg` for generating graphs [55]. We wish the SSA format was documented in greater detail on this page.

We found the CFG dump inferior to the SSA dump because it does not explicitly state the edges. As for the VCG dump, we found that the VCG format comes from an early compiler pass that can very rarely contain dead code. For sharing the structural data of sensitive code with other parties, this could prove to be a useful feature. For our purposes, the dead code was not worth it.

The compiler throws out nonfunctional lines. For instance, if a function call taking many arguments is spread across multiple lines, the compiler will treat the call as if it were on only one line. This behavior is problematic for our purposes. To resolve this issue, we let lines that the compiler does not specify inherit the last known testability.

7.3.3. **Vision**

Our vision is to use *some* fault proneness metric to guide automatic code refactoring. In this section, we give an example using testability for illustration purposes. We have not yet performed a longitudinal study validating this particular vision, but it is in the spirit of McCabe's ideas about “reducing” code [25].

Figure 1 contains part of the output from our Java tool: the testability of a 12 node control flow graph (the mapping from nodes to source lines is omitted).

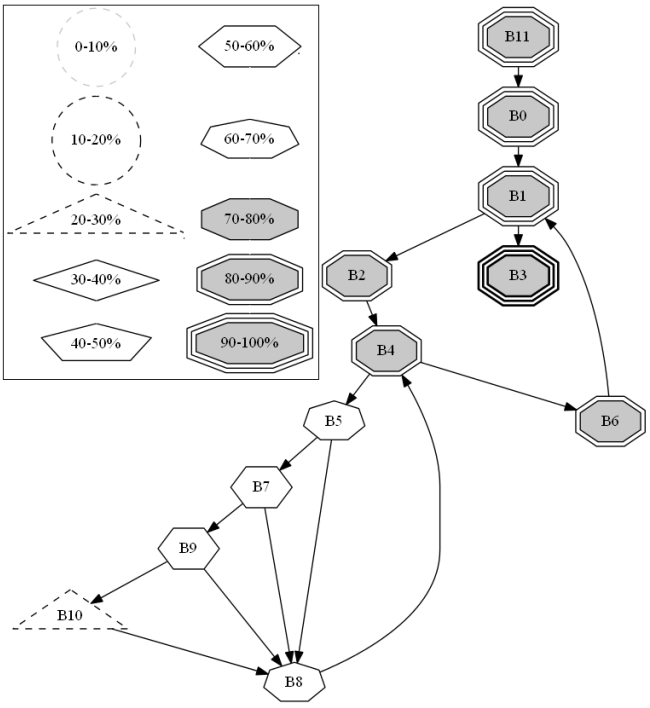

Figure 7.1. A function's testability, before refactor.

As Figure 1 shows, node B10 is hardest to test. We envision an automatic refactoring process that would allow us to fold B10 and B9 together by moving that code into its own function.

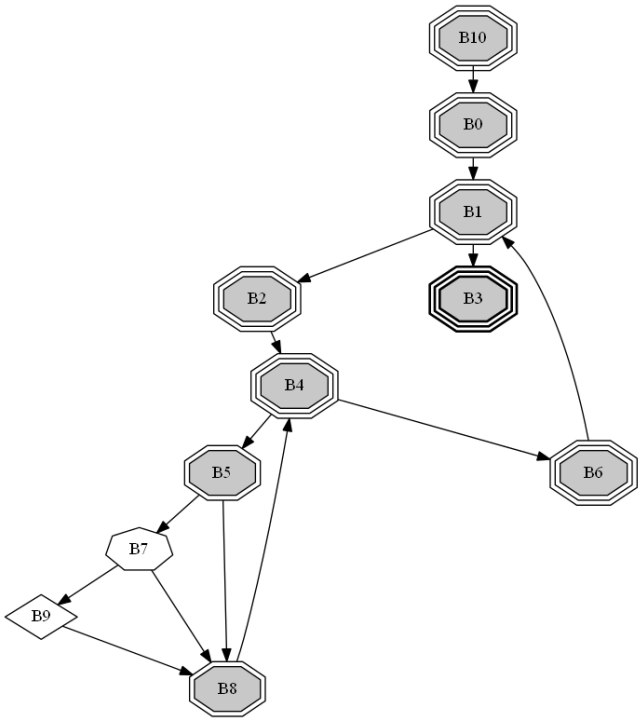

Figure 7.2. A function's testability, after refactor

Intuitively, as Figure 2 shows, this change would reduce the number of test paths and thus make the code overall easier to test. It should be very feasible to implement this process with a per-line metric such as testability because it precisely identifies fault-prone code. We posit that this process could reduce the fault proneness of the code by making the code less complex.

7.4. Case Study

In our case study, we examined the link between testability and fault proneness in Apache HTTP Server, otherwise known as "`httpd`" or simply "Apache." We also examined the quality of testability as a fault proneness predictor in terms of recall and precision. To our knowledge, no one has studied fault proneness on a per-line basis, so for comparison, we also compared two other metrics: McCabe cyclomatic complexity and random.

Apache is a well-known open-source HTTP server written entirely in C. Its licensing allows many commercial HTTP servers to reuse its code. We examined the Apache "trunk" SVN revision history to conduct our case study.

7.4.1. **Research Questions**

This section states our hypotheses. We were interested in two ideas: RQ1) how the testability of Fault Prone (**FP**) code compared with Not Fault Prone (**NFP**) code, and RQ2) how well our testability measure predicted the precise location of faults.

Difference in means (RQ1)

Our null hypothesis (H_0) states that there is no difference between the mean testability of FP code and the mean testability of NFP code. Our alternate hypotheses states that there is a difference, either:

- H_1: FP code had lower testability than NFP code (what we posit), or
- H_2: FP had higher testability than NFP code.

We reject the null hypothesis if the difference is significant within 95% confidence ($\alpha = 0.05$). We accept the null hypothesis only if we have at least 80% statistical power ($\beta = 0.2$) and the difference is not significant.

Recall and precision (RQ2)

To determine the quality of our predictions, we also performed a traditional fault proneness experiment by assessing the recall and precision of our estimates. We define recall, precision, and the hybrid measure F1 as:

$$\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} \tag{3}$$

$$\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}} \tag{4}$$

$$\text{F1} = 2\,\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}. \tag{5}$$

Thus our null hypothesis is that there is no difference in recall and precision between testability, McCabe cyclomatic complexity, and random. Our alternate hypothesis is that there is a difference (two-sided).

7.4.2. **Procedure**

We examined 43 random revisions modifying C code of Apache trunk in the range [1082630,1377685]. We picked 43 based on a preliminary statistical power analysis for an overall hypothesis test. We skipped revisions correcting spelling mistakes in comments or Windows-specific code (we could not cross-compile it on our Linux system). When we picked revisions at random from the above range, we noticed most of them changed the Apache "modules" as opposed to the actual server. Historically, we observed the proportion in Apache was more even. To reduce the potential bias of using so many revisions to modules, we selected 22 revisions to the modules and 21 revisions to the server.

For each revision, we identified the C files changed by that revision. For each changed C file, we ran our tool to compute the testability of each source line. The testability scores for the changed lines formed the FP data set. The remaining scores went into the NFP data set.

To implement this procedure, we configured the SVN diff to output an `ed` [14] script, an old but easy-to-parse form of diff that gives the line numbers of changed and deleted lines with respect to the original file's line numbers. We then built the database for the original revision and queried the database for the testability scores of the changed lines. To handle added lines, we computed the testability of the added lines in reverse: we updated to the next revision, recompiled the database, and extracted the deleted line numbers in the reverse SVN diff.

Testability returns a number in [0,1], but cyclomatic complexity returns a positive integer; both metrics need thresholds defining whether a line is FP or NFP. To set thresholds in a "fair" way, we computed 11 percent ranks in [0,1], incrementing 0.1 every time. For each rank, we computed the corresponding testability and McCabe cyclomatic complexity. We used the values as our FP/NFP thresholds.

We also plotted the recall and precision from randomly ranking blocks. In theory, the recall of random choices should scale linearly from 0 to 1, while the precision of random should remain roughly flat at the overall FP sample proportion. On the graphs of recall, precision, and F1, methods that are better than random will be superlinear (above random) while methods that are worse than random will be sublinear (below random).

7.4.3. **Results**

This section discusses the results of the comparison of testability means as well as the recall and precision.

Difference in means (RQ1)

Table I summarizes the files present in the 43 revisions.

Table 7.1. Apache summary statistics

Measure	FP	NFP
Basic blocks	341	22638
Mean	**0.34**	**0.31**
Variance	0.10	0.12

As Table I shows, only about ~1.5% of basic blocks changed over the 43 revisions. The FP code, on average, had higher testability than the NFP code. In other words: developers found faults in easy-to-test code, supporting H_2. The data was not normally distributed, so to determine the significance of the difference in the means, we used the `ranksum` test in Matlab corresponding to the Mann-Whitney U-test nonparametric comparison of means.

Table II gives the results of the test.

Table 7.2. Results of U-test between FP and NFP groups

Measure	Value
p-value	$1.35 * 10^{-6}$
Effect size (Cohen's d)	0.06

As Table II shows, the difference in the means was significantly different and exceeded our confidence threshold of 95%. The effect size, Cohen's *d*, (the difference between means normalized by a standard deviation) was only 0.06, but the difference was still significant because the data was not normally distributed. Thus we reject H_0 in favor of H_2.

To perform finer-grained testing, we applied k-means to cluster the blocks into three testability groups: low, medium, and high testability. Within each cluster, we again partitioned the blocks into FP and NFP sub-clusters and repeated our analysis. We applied the t-test to compare the FP and NFP sub-clusters. Table III summarizes our results.

Table 7.3. Results of 3-means clustering

Measure	Low	Medium	High
Blocks (FP)	209	72	60
Blocks (NFP)	14157	4139	4342
Mean	0.0780	0.4487	0.9462
Mean (FP)	**0.1215**	0.4586	0.9394
Mean (NFP)	**0.0772**	0.4486	0.9463
Variance (FP)	0.0048	0.0130	0.0073
Variance (NFP)	0.0048	0.0161	0.0081
p-value	$4*10^{-17}$	0.4611	0.3336
Effect size d	**0.6428**	0.0788	0.1204
Statistical power	100.00%	**9%**	**15%**

Within the "Low" testability cluster, the FP blocks had significantly higher testability that the NFP blocks. The effect size was moderate at 0.6428. Thus the trend of developers finding bugs in easy-to-test code was actually strongest in the Low group.

The medium and high testability clusters are inconclusive. The corresponding p-values in Table III lead us to not reject H_0. However, the statistical power is too low to actually accept H_0. Increasing statistical power is tricky because it is impeded by unequal sample proportions. While we could have easily looked at more revisions, we cannot convince the Apache developers to change more lines of code per revision.

Recall and precision (RQ2)

Figures 3, 4, and 5 show the recall, precision, and F1 of predicting the precise location of faults using the same 43 revisions from the first part of the study. The "Random-" plots show the effects of picking the stated percent of blocks at random. The "Testability-" plots show the effects of using the percent rank of the testability scores as an upper bound threshold. The "McCabe-" plots similarly show the effects of using the percent ranks of the cyclomatic complexity as a lower bound threshold (we order this series in reverse for comparison).

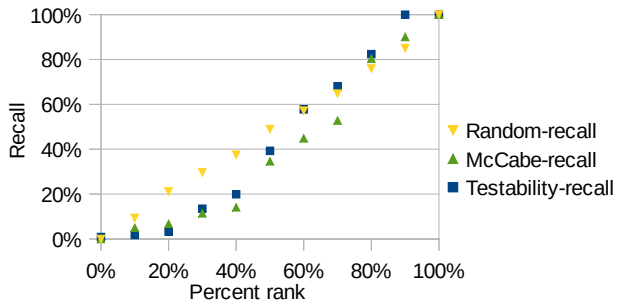

Figure 7.3. Recall of testability, cyclomatic complexity, and random

As Figure 3 shows, testability was not significantly different than random at recall. According to a matched pair t-test, the p-value was 0.16. McCabe had significantly worse recall than testability (p-value 0.04) and random (0.01).

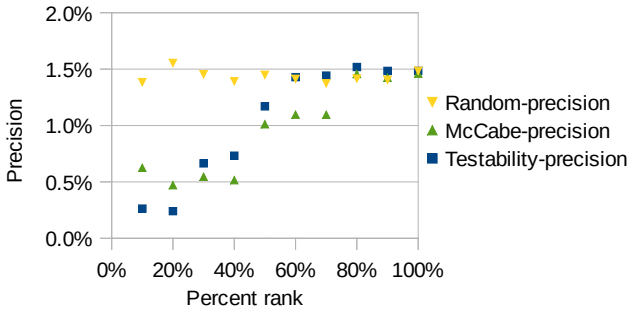

Figure 7.4. Precision of testability, cyclomatic complexity, and random

As Figure 4 shows, testability did not have significantly different precision than McCabe (p-value 0.49) despite appearing slightly higher. Both metrics had significantly worse precision than random (p-values 0.048 and 0.006, respectively).

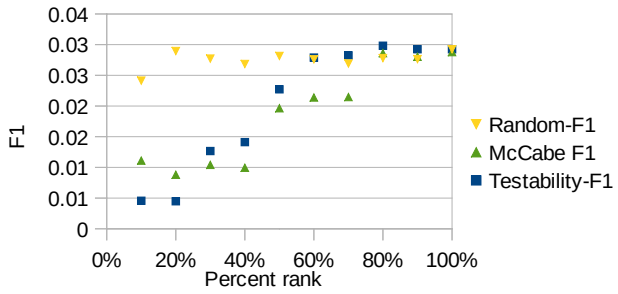

Figure 7.5. F1 of testability, cyclomatic complexity, and random

As Figure 5 shows, testability did not have significantly higher F1 than McCabe (p-value 0.15), but had significantly worse F1 than random (p-value 0.04). McCabe was also significantly worse than random (p-value 0.03). The F1 for both methods peaked at the 80% percent rank. This rank corresponds to a cyclomatic complexity of 10 or more, which was McCabe's rule of thumb [25]. The testability at the 80% rank is 0.66, suggesting a rule of thumb that code with testability less than 0.66 needs testing.

In absolute terms, the precision of all methods examined was atrocious. Although testability significantly improved recall over McCabe, neither method was especially precise. This point leads into our threats to validity.

7.4.4. **Threats to validity**

In terms of *content validity*, we did not study all revisions. While we could have studied more revisions, we could not study *all* past revisions because the older revisions require significant system modification to compile. This factor artificially caps the precision of random. Even if we could compile all revisions, the data would still be incomplete for the usual reason: the false positive code could indeed have faults that have yet to be discovered. Thus the true precision of the methods could be understated. Mitigating this threat is the fact that the data still supported McCabe's rule of thumb.

In terms of *internal validity*, there could be selection bias from using recent revisions. We tried to mitigate selection bias by being random, but a random selection across a longer time period would have been better. As said above, we could not compile very old revisions. Also, we did not model all possible effects on testability, for instance the effects of specific files or effects of the modules and server directories; we treated these as random effects.

In terms of *external validity*, we only studied Apache. While Apache is a very "real world" project, it could be that other projects, such as Eclipse, display different trends regarding testability and cyclomatic complexity. Hata et al. note that fault proneness metrics tend to perform especially well on Eclipse [24], so our results might not be directly comparable with Eclipse papers.

7.5. Conclusion and Future Work

We examined recent revisions in the Apache SVN log and found, to our surprise, that developers had a tendency to find bugs in easy-to-test code. The more complicated the code became, the

more evident the pattern became. This result went against our posited hypothesis that developers find bugs in hard-to-test code.

In Apache, we found the recall of testability as a fault-proneness predictor was significantly better than McCabe cyclomatic complexity when applied on a per-line basis. We found evidence confirming McCabe's “10-or-more” rule of thumb for deciding what code to test. We introduced our own rule of thumb: lines with testability 0.66 or less need to be tested.

Our challenge to the fault proneness community is to use fault proneness metrics to make specific code improvement recommendations. We hope that such recommendations will help practitioners improve fault-prone code and thus simplify their testing efforts. We discussed one possible approach: using a line-specific fault proneness metric to automatically refactor fault-prone lines out of complicated functions. We constructed testability with this approach in mind. The current function-level metrics we see in fault proneness studies, such as McCabe cyclomatic complexity, do not suffice for our purposes.

Chapter 8. A Framework for Assessing the Validity of Mutation-based Experiments

M. Hays and J. H. Hayes, “A Framework for Assessing the Validity of Mutation-based Experiments,” University of Kentucky, Lexington, KY, Technical Report 530-14.

8.1. Summary

Software testing researchers frequently must empirically demonstrate that new testing techniques can be used to find faults. Researchers usually construct controlled experiments by seeding known faults into programs under test. Program mutation is a useful tool for generating faults in situations when it is either too expensive or impossible to retrieve large numbers of faults from the programs' history. However, it is not known for certain whether such mutations are materially similar to naturally occurring faults. To facilitate this comparison, we present an experiment framework to assist the mutation testing community to establish whether faults generated by program mutation fail as often as real-world faults. We introduce a probabilistic definition for estimating a mutation's failure rate. We decompose the definition into several events and demonstrate ways to statically measure the likelihood of each event. We provide worked examples with systematically computed definitions for all observed mutations in a small program, TriTyp.

8.2. Introduction

Program mutation has many uses to both researchers and practitioners. Practitioners can use mutation to boost the sufficiency of their existing test sets. Software testing researchers can use mutation as a source of faults for their empirical validation of new ways to test.

One open problem hindering the wider adoption of mutation is that it can be very time-consuming to produce a test set that kills all mutants. Worse, sometimes equivalent mutants arise that do not actually alter the functionality of the program, requiring further analysis. To reduce the overall effort involved in building these test sets, much mutation research has focused on the determination of sufficiently small sets of mutations. Usually success is indicated when the resulting test set can find excluded mutants as well.

Another open problem is whether mutants are suitable substitutes for real faults; this is called the coupling effect hypothesis [18]. It has been demonstrated that the mutant coupling holds: test sets that can identify simple mutants can generalize to more complicated pairs of mutants [18]. While this result is encouraging, a more robust framework is needed to confirm the generalized coupling hypothesis.

In this problem description report, we make several contributions toward understanding these issues. We introduce a probabilistic definition to express existing concepts such as *testability* and *fault size*. We split this definition into several components that can easily be determined for mutants and discuss approximations for naturally occurring faults. We provide a worked example showing how one would build a fault size profile for a program processed by the mutation tool MuJava [56]. We then describe an experiment framework for determining the validity of the use of mutation in software testing experiments. We also discuss potential ramifications of our work to mutation testing research.

This paper is organized as follows. Section III outlines our formal model. Section IV provides worked examples from the TriTyp program. Section V outlines our validation framework applied to mutation testing. Threats to validity are presented in Section VI. Section VII discusses planned future work.

8.3. Definitions

In this section, we describe our formal model of fault size. We also list the MuJava mutant classes referenced in this paper.

8.3.1. **Fault size**

Voas's testability and our take on fault size are similar. As Voas [27] originally outlined, there are three events that occur when a test fails. Let F be the event that a test suite experiences at least one failure due to a single fault. We formally denote the events leading up to F using the PIE notation:

E: the code containing the fault is executed.

I: the fault corrupts the internal state.

P: the program propagates the bad internal state to the test case.

Equation 1 formally describes the relation of these events:

$$\Pr(F) = \Pr(E \cap I \cap P). \quad (1)$$

We can rewrite this as:

$$\Pr(F) = \Pr(E)\Pr(I|E)\Pr(P|I \cap E). \quad (2)$$

Note that Voas also factored in the input distribution into his definition of testability. We now provide definitions for each of these components.

8.3.2. **Testability**

In our previous work [26], we established that the likelihood of code being executed can be statically estimated by path analysis of the faulty code. Li et al. at George Mason University [31] provided us with their tool to produce a minimal set of test paths to establish code coverage. We originally made the *simple fault assumption* that a fault can only exist in a single block. Given that a fault is located in basic block *b* of a function *f*, its likelihood of being executed is:

$$\Pr(E(b)) = \frac{\#\ paths\ in\ f\ containing\ b}{\#\ paths\ in\ f}. \tag{3}$$

There are some cases where MuJava can generate faults that span multiple blocks, but in those cases, we observe that it is sufficient to execute any of those blocks to notice the mutant. For naturally occurring faults that are localizable to some number of blocks, the answer is less clear. We offer alternative formulations of fault size in subsequent sections that work around this issue.

8.3.3. **Propagation**

To induce a failure, it is not enough for a mutation to be executed. The result must also propagate to the output. Santelices and Harrold [30] described a very simple formula to represent this situation. If an affected definition *d* is accessible to the test, then they say the error *always* propagates. Otherwise, if the code branches *N* ways, then the fault propagates when the code chooses one of the *N*-1 wrong branches. Santelices and Harrold implicitly assume that each branch is equally likely to be executed with probability $\frac{1}{N}$:

$$Pr(P|I \cap E) = \begin{cases} 1 & \text{if } d \text{ is accessible} \\ \frac{N-1}{N} & \text{otherwise} \end{cases}. \tag{4}$$

We suggest tweaking (4) by letting *N* count the number of *paths* leaving *d*. This tweak more intuitively represents how faults unpredictably impact the overall program flow.

8.3.4. **Intrinsic fault size**

The most important probability, Pr(*I* | *E*), can be referred to as a fault's *intrinsic size*. The intrinsic size measures how likely it is that a fault will corrupt the internal data state, regardless of where it is inserted into the code.

Mutations

In the context of mutation, it makes sense that a given mutation would have some intrinsic difficulty associated with it, regardless of where the mutation is applied in the program. Seeing as many mutations take arithmetic expressions as arguments, we offer a definition relevant to those

mutations. Let p be the un-mutated expression. Let p' be the mutated expression. Let X represent the possible variable assignments in p. The fault's intrinsic size is:

$$\Pr(I|E) = \sum_{x \in D} \Pr(X = x) * (p(x) \neq p'(x)). \quad (5)$$

Generally, we know the data types in the program domain (such as integers, floats, strings, etc.) and the values those types take, but we don't know the distribution of the values. This complicates the evaluation of $\Pr(X = x)$. In this typical case, the principle of maximum entropy suggests that the uniform random distribution is statistically sufficient:

$$\Pr(X = x) = \begin{cases} \frac{1}{|D|} & \text{if } x \in D \\ 0 & \text{otherwise} \end{cases}. \quad (6)$$

Note that these intrinsic size definitions apply to MuJava traditional mutations only.

Naturally occurring faults

Unlike mutations, naturally occurring faults can be syntactically very large, so attempting to precisely frame them as above can be tricky. If we have a representative test set and assume that a single fault is present, we can approximate Pr(*F*) by the proportion of failing test cases in a program's test set. This lets us use (2) to evaluate intrinsic fault size in the following way:

$$\Pr(I|E) = \frac{\%\text{failures}}{\Pr(E)\Pr(P|I \cap E)}. \quad (7)$$

In practice, we can reduce the expression $\frac{\%\text{failures}}{\Pr(E)}$ by counting only the proportion of failing test cases that actually executed the fault. Furthermore, if there was at least one test failure, we can posit that $\Pr(P|I \cap E) = 1$ because the propagation of the fault is no longer an uncertainty. Thus we have:

$$\Pr(I|E) = \frac{\%\text{failures caused by F}}{\%\text{test cases executing F}}. \quad (8)$$

The program instrumentation required to make this determination is very similar to that used in spectrum-based fault localization: one must associate the code coverage of each test case with the test's result. Under these conditions, one can determine the intrinsic size of arbitrary naturally occurring faults.

8.3.5. **Mutation classes**

In the next sections, we refer to several abbreviations for mutant classes. For reference, these are [57]:

- **AOIS:** Arithmetic Operator Insertion - Shortcut
- **AOIU**: Arithmetic Operator Insertion - Unary
- **AORB**: Arithmetic Operator Replacement Binary
- **COI**: Conditional Operator Insertion
- **COR**: Conditional Operator Replacement
- **LOI**: Logical Operator Insertion
- **ROR**: Relational Operator Replacement.

According to MuJava, there are 15 method-level mutations in total. A program that covers the unlisted mutations must use arithmetic shortcuts, perform bitwise Boolean arithmetic (both unary and binary), and perform bit shifts. Arithmetic shortcut mutations are equivalent to AORB mutations because they mutate the same operators. The unary deletion operators are equivalent to their insertion operators from a fault size perspective; the order doesn't matter. Bitwise operations are harder to find, but we can probably ignore these mutations without loss of generality to our results in most programs.

We've posited that the intrinsic fault size of each mutation is constant for a given input profile. While a diverse class of mutations, such as ROR, may contain many mutations, each mutation within that class is allowed to have its own intrinsic fault size. This view of mutations facilitates the following worked examples.

8.4. Worked Examples

In this section, we describe how to apply our definitions of fault size to MuJava mutants generated for the TriTyp [58] program. We characterize the resulting distribution of fault sizes, then show worked examples for computing sample fault sizes from each mutant class.

8.4.1. **Approach**

We ran MuJava on TriTyp. For each mutant version of TriTyp, we determined the specific characters changed based on the diff [59] of the source code. From this, we derived the mutation being applied and computed its intrinsic size. MuJava applies the same mutation at different points of the Trityp code, so we constructed a feedback-driven tool to help speed up the classification process. We represent a mutation as a quadruple listing the number of values in the mutation, the operators changed, and the mutation's intrinsic size:

(#tokens, before_operators, after_operators, intrinsic size)

We keep a lookup table of these quadruples. As our tool scans the mutated source code, it performs a diff to identify the changed line, then tries to match the diff to entries in the lookup table. When a match is found, it reports the match as a suggestion to the analyst.

For instance, consider the diff in Listing 1 of the following ROR mutant, ROR_1:

```
34c34
<          if (Side1 <= 0 || Side2 <= 0 || Side3 <= 0) {
---
>          if (Side1 > 0 || Side2 <= 0 || Side3 <= 0) {
```

Listing 1. Diff of Trityp between original code and mutation ROR_1.

As the diff shows, ROR_1 replaces the les-than-or-equal operator with a greater-than operator. It involves two values: Side1 and 0. When the tool first encounters this mutation, it prompts the analyst to specify the intrinsic size. As we will later demonstrate, the intrinsic fault size of this mutation is very large; it is in fact 1.0. The tool records this information in the following quadruple:

(2,{<=},{>},1.0)

When the tool comes across future instances of this mutation, it automatically suggests this intrinsic size to the analyst. Higher-order mutations that involve many values take precedence in this recommendation procedure.

8.4.2. **Summary of results**

Table 1 present the results of our calculations after applying MuJava to the TriTyp program and computing the intrinsic fault size of each mutant. For brevity's sake, we recorded those mutants whose sizes were approximately zero (less than 10^{-6}) as zero. We similarly recorded mutants that were approximately one as one. As there are many mutants, we list summary statistics by mutant type.

Table 8.1. Intrinsic Fault sizes for TriTyp mutants

Mutant Class	# Mutants	Mean	Variance
AOIS	136	1	0
AOIU	7	1	0
AORB	36	1	0
COI	24	1	0
COR	14	0.48	0.03
LOI	39	1	0
ROR	85	0.6	0.1

The AOIS AOIU, AORB, COI, and LOI classes of mutations were consistently very large. The COR and ROR classes had some slight variation but were medium sized on average.

Figure 1 summarizes the total distribution of fault sizes:

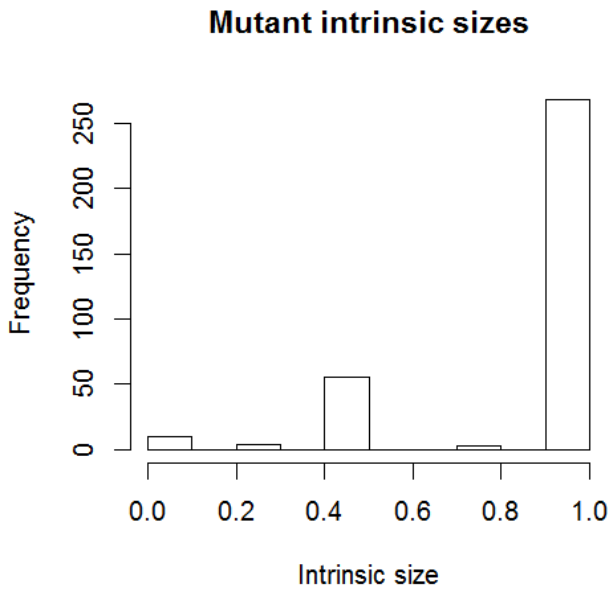

Figure 8.1. Intrinsic size of TritTyp mutations.

As Figure 1 shows, the mutations overall tended to be very large. There were a few very small faults. This would suggest that most Trityp mutants are easily killed. In the next subsections, we provide some worked examples demonstrating how we arrived at this distribution.

8.4.3. **Arithmetic Operator Insertion - Shortcut**

The AOIS class of mutations inserted increment (++) and decrement (--) operators on all numeric variable uses. For instance, the mutation AOIS_1 replaced a reference to an integer input, Side1, with ++Side1. As Side1 can be any integer, we define the domain to be the set of integers, Z. We formally derive the intrinsic size of this mutation as follows:

$$\Pr(I|E) = \sum_{x \in Z} \Pr(Size1 = x) * (x \neq x + 1) \quad (9)$$

$$= \frac{1}{|Z|} \sum_{x \in Z} (x \neq x + 1) \quad (10)$$

$$= 1. \quad (11)$$

8.4.4. **Arithmetic Operation Insertion - Unary**

The AOIU class negated numbers. For instance, AOIU_1 negated a variable called triOut, which is the returned classification. The negation operation essentially affects every Integer except zero. It could be argued that triOut is only allowed to take a few values over the course of execution and thus the domain Z is not an appropriate fit. Clearly though this constraint would materially not affect the following evaluation.

$$\Pr(I|E) = \sum_{x \in Z} \Pr(triOut = x) * (x \neq -x) \quad (12)$$

$$= \frac{1}{|Z|} \sum_{x \in Z} (x \neq -x) \quad (13)$$

$$= \frac{|Z|-1}{|Z|} \approx 1. \quad (14)$$

8.4.5. **Arithmetic Operator Replacement - Binary**

The AORB class replaced binary arithmetic operations such as addition, subtraction, multiplication, division, and bitwise operations, with one another. For instance, AORB_1 replaces the sum of triOut+1 with the product triOut*1. Clearly every execution is affected.

$$\Pr(I|E) = \sum_{x \in Z} \Pr(triOut = x) * (x + 1 \neq x * 1) \quad (15)$$

$$= 1. \quad (16)$$

8.4.6. **Conditional Operator Insertation**

The COI class negated logical expressions. For instance, AORB_1 negates the expression Side1<=0. The resulting intrinsic size formula is a tautology.

$$\Pr(I|E) = \sum_{x \in Z} \Pr(Side1 = x) * (x \leq 0 \ \neq !\,(x \leq 0)) \quad (17)$$

$$= 1. \tag{18}$$

8.4.7. **Conditional Operator Replacement**

The COR class replaced binary Boolean operators, such as OR (||), AND (&&), XOR, etc. For instance, COR_1 replaces Side1 <= 0 || Side2<=0 with Side1<=0 && Side2<=0. Examining the truth table, we see that this replacement will affect half of all possible return values of the underlying predicates.

$$\Pr(I|E) = \sum_{x \in Z} \Pr\begin{pmatrix} Side1 \leq 0 = x \ \cap \\ Side2 \leq 0 = y \end{pmatrix} \tag{19}$$
$$* (x \,||y \ \neq x\&\&y)$$

$$= 0.5. \tag{20}$$

This category had some slight variance because different replacements have different effects. For instance, replacing OR with XOR will only affect 25% of the truth table, while replacing AND with XOR will affect 75% of the table.

8.4.8. **Logical Operator Insertion**

The LOI class computed the 1's complement of numbers. For instance, LOI_1 replaces Side1 with ~Side1. There is a 1-1 mapping of numbers and their 1's complements and none of them map to themselves, so every value will be drastically affected:

$$\Pr(I|E) = \sum_{x \in Z} \Pr(Side1 = x) * (x \neq \sim x). \tag{21}$$

8.4.9. **Relational Operator Replacement**

The ROR class replaces binary Boolean operators, such as less than, greater than, etc., with one another. For instance, ROR_1 replaces Side1<=0 with Side1>0. In this case, the intrinsic size formula is a tautology.

$$\Pr(I|E) = \sum_{x \in Z} \Pr(Side1 = x\,) * \tag{22}$$
$$(x \leq 0 \neq x > 0)$$

$$= 1. \tag{23}$$

The ROR class possessed the most variance in intrinsic size of all of the mutant classes we studied. For instance, ROR_3 replaces Side1<=0 with Side1<0. Under our usual assumptions, there is a miniscule chance that Side1=0. Thus, the intrinsic size of ROR_3 is close to zero - the complete opposite of ROR_1. This variance makes the ROR mutants the most diverse of the mutations applicable to Trityp.

8.5. Validation Framework

With these definitions and worked examples in mind, we outline our framework for mutation researchers:

1. Given a program with a known test set and fault set, generate mutants for the program.
2. Determine the distribution of the mutant faults' intrinsic sizes.
3. Determine the distribution of the real faults' intrinsic sizes.
4. Using the appropriate statistics, determine whether the two distributions are significantly different.

There are several competing hypotheses:

H0: mutants have the same fault size as naturally occurring faults.

H1: mutants are significantly smaller than naturally occurring faults.

H2: mutants are significantly larger than naturally occurring faults.

H0 suggests that mutants and naturally occurring faults are equally useful for software testing research. H1 might help explain why Offutt's coupling hypothesis [18] holds; if a test set can find intrinsically small mutant faults, then surely it can find intrinsically large naturally occurring faults. If H2 were to hold, then it would suggest that mutation boils down to making sure that the test set is covering the code.

8.6. Threats to Validity

Our calculation of the intrinsic fault size of MuJava mutants of TriTyp and subsequent analysis of the mutation operators is subject to a number of threats to validity.

8.6.1. **Conclusion validity**

A possible conclusion validity threat is that of *reliability of treatment implementation.* This threat arises due to our implementing a tool to calculate the mutation operators applied by MuJava, a basis for the rest of the work. If the tool is not correct, the entire study could be flawed. We feel this risk is fairly minimal as the algorithms required are fairly simple, table look ups, and we tested the tool extensively. A possible threat to conclusion validity is that of *reliability of measures*. It is possible that our definition of intrinsic size is not correct; this could lead to flawed sizes for the mutation operators and could lead to incorrect conjectures about the frequency and/or defect detection effectiveness of the operators. This threat can be mitigated by

examining other definitions of intrinsic size, repeating the study with those definitions, and comparing the results.

8.6.2. **Internal validity**

A possible threat to internal validity is that of *ambiguity about direction of causal influence*. Is it the case that intrinsic fault size is "causing" defect detection effectiveness, or vice versa?

8.6.3. **Construct validity**

There are two possible threats to construct validity. First, it is possible that we have *mono-operation bias*. Our study is conducted with a single program as object, so the cause construct is under-represented. Second, it is possible that we have *mono-method bias*. As we use just one type of measurement to estimate the intrinsic fault size, we may have a measurement bias. To mitigate this, we can use multiple measures and compare (as in reliability of measures above).

8.6.4. **External Validity**

There are two possible threats to external validity. It is possible that we have *interaction of selection and treatment.* Our object of study, TriTyp, represents but one program in one domain. Our results cannot be generalized to other programs or domains or programming languages. In order to mitigate this threat, the study must be replicated on numerous, diverse programs. It is possible that we have *interaction of setting and treatment.* Some may consider TriTyp to be a "toy" program as it is quite small. However, it has been used in many testing and mutation testing papers and has been studied extensively. Also, it is well known that even small programs generate a large number of mutants. Studying a large program, therefore, is problematic.

8.7. Conclusions and Future Work

Our results open many interesting possibilities that warrant investigation. We found that the mutant classes AOIS, AOIU, AORB, COI, and LOI were all intrinsically large. According to our model of test failure, practices such as code coverage and design for testability would guarantee the death of these mutants. To improve the test set itself, we hypothesize that it would be more productive to apply mutations from smaller classes such as COR or ROR, then use a small selection from the larger classes to ensure every definition is mutated at least once. This is a readily testable hypothesis for researchers working in the area of developing sufficient sets of mutants.

Our model of test failures could help identify equivalent mutants early in the testing process. While we did not come across any mutants in TriTyp that truly had zero intrinsic fault size, if we were to, clearly these would be equivalent mutants. Changes that cannot propagate also yield equivalent mutants, but the Santelices and Harrold definition of propagation [30] always allows for *some* chance of propagation.

MuJava supports several mutations that could be applied to TriTyp. While the deletion mutations will simply mirror their insertion counterparts, it would be interesting to investigate intrinsic fault size of the bitwise mutations. This would give us a clearer idea of the complete mutant fault size distribution. With this type of distribution available, we intend to use the outlined framework to validate the applicability of mutation in software testing experiments. Completing this step will allow us to say definitively whether mutants are materially similar to naturally occurring faults.

Chapter 9. Statistical Analysis for Traceability Experiments

9.1. Summary

An important aspect of traceability experiments is the ability to compare techniques. In order to assure proper comparison, it is necessary to perform statistical analysis of the dependent variables collected from technique application. Currently, there is a lack of components in TraceLab to support such analysis. The Software Verification and Validation Research Laboratory (SVVRL) and the Statistics Department of the University of Kentucky have developed a collection of such components as well as a workflow for determining what type of analysis to apply (parametric, non-parametric). The components use industry-accepted R algorithms. The components have been validated using independent standard statistical algorithms applied to publicly available datasets. This work addresses the Purposed grand challenge (research project 4) and Cost-Effective Grand Challenge (research project 4) as well as the Valued Grand Challenge - research project 6.

9.2. Introduction

Early traceability papers rarely applied statistical analyses as the authors were only able to examine two or three datasets and knew that such a small sample could not lead to statistically significant results. With the advent of the use of Mean Average Precision (MAP) and other “per query” measures, traceability researchers now have many more data points (a dataset that has 50 queries searching into 150 elements now has at least 50 data points versus being considered one dataset). With these larger sample sizes, it is now incumbent on traceability researchers to apply statistical analyses to the dependent variables they collect when running experiments.

This leads to the next conundrum. What statistical techniques should be used? How can traceability researchers overcome the parade of criticism from reviewers such as: "your data did not conform to the assumptions of the statistical technique used,” your test did not have sufficient power,” and/or “you cannot use the mean with that type of data.”

This Challenge paper seeks to address some of the aforementioned concerns by providing a collection of TraceLab components that take the dependent variables from experiments (such as MAP, F, recall, precision) and determine what tests are required, check the appropriate assumptions, and run the tests. This paper contains standard language that can be used in traceability papers to demonstrate to reviewers that proper statistical analysis, designed by statisticians, has been applied.

The paper is organized as follows. Section 2 discusses statistical tests for traceability. Section 3 presents some thoughts on statistical testing. Section 4 discusses the TraceLab components developed for statistical analysis. The standard language to be used in papers employing these Statistics components is provided in Section 5. Section 6 discusses evaluation of the TraceLab statistic components, and Section 7 concludes and discusses future work.

9.3. Statistical tests for traceability

Currently, it is becoming more commonplace to see non-parametric techniques applied to dependent variables (such as MAP) in various experiments. Examples include Kong, Hayes, Dekhtyar, and Dekhtyar (used Wilcoxon Signed Rank test) [60], Niu and Mahmoud (used Mann Whitney) [61], and Shin, Hayes, and Huang (examined correlation of commonly used measures and their analysis) [62].

It is rare for parametric tests such as student's t or ANOVA to be applied. It appears that this is due to author fear of reviewer criticism versus due to data not meeting required assumptions (such as normality). Yet when normality and equal variance assumptions are met, appropriately chosen parametric tests are more powerful than their non-parametric counterparts and thus should be considered first. Our TraceLab components support such consideration, making statistics accessible to all researchers, even those who may not feel comfortable working with statistics.

9.4. Statistical tests

Selecting an appropriate inferential method for statistical analysis is a complex and highly interactive task. Typically, there is not one correct procedure, but there are some that are more appropriate and others less and some simply inappropriate. An expert statistician will consult diagnostic plots, test statistics, p-values, and transformations, among other tools, in order to choose a method that is adequate and powerful. Automating the process of test selection may therefore draw criticism: no automated procedure will be able to substitute expertise and experience. On the other hand, with widespread availability of free statistical software packages, the application of statistical procedures is at the fingertips of many. Many researchers simply don't have advanced statistical expertise or experience, or even quick access to expert statistics knowledge to choose the most appropriate method, or to decide when a standard method is not appropriate. The MeansTest algorithm will be useful for this group of researchers. It is designed to imitate the major decisions a statistician would make when analyzing two-sample data.

Indeed, the first decision is whether the two samples are independent or paired. If paired, then for normal data, the paired t-test [38] is the method of choice, while the signed rank test [39] is its nonparametric alternate. For independent samples, even more important than checking normality is whether it is reasonable to assume that both samples come from distributions with equal variances. If not, there exist powerful approximate methods for the normal distribution case [40], [63], [64], and for the case in which normal distributions cannot be assumed (the Brunner-Munzel test [40] and its permutation version).

The latter is also an example that the statistics research community continues to derive and validate new and more powerful or more robust inferential procedures, so that updates on the decision trees may have to be made. For example, for the comparison of two independent samples of non-normal data, the rank-sum test [64] has been the method of choice for several decades. However, it assumes that under null hypotheses, the variances of both samples are equal. Just recently, the Brunner-Munzel [40] test has been devised and validated to provide a nonparametric test for location in the presence of unequal variances. Even more recently, the performance of this test has been improved by using a permutation approach.

How are the decisions regarding normality and unequal variances made? Normality can be assessed using the Shapiro-Wilk [42] test. However, since the t-test is rather robust against violations of the normality assumption, an alpha-level of 5% can be chosen as a threshold. In the case of two independent samples, neither should show strong evidence of non-normality. The assumption of equal variances in the case of two independent samples is rather important and is tested using the Levene-Brown-Forsythe [65] test at the 5% level.

9.5. TraceLab Statistical Components

We implemented all of the above tests as individual TraceLab components. In this section, we describe the implementation details of our composite component, MeansTest, as well as our experiences with TraceLab.

9.5.1. R Implementation

It is straightforward to calculate many test statistics, such as the t statistic for the t-test. However, most researchers are interested in the p-value of the test statistic, which expresses the significance of the result. The computation generally does not have an easily computed closed form, so implementing this step by hand is undesirable. TraceLab already links with the commercial

library ALGLIB [66] that provides a limited selection of hypothesis tests that return p values. R [44], a popular statistics language, supports several tests not in ALGLIB that are relevant to our research. In the interest of maximum code reuse, we wrote our statistics in R.

We wrote a TraceLab helper component, called Rscript, which returns an opaque reference representing the R runtime. TraceLab components can use this opaque reference to execute any R script. In a TraceLab experiment, the user simply specifies the path to his or her Rscript.exe in the helper's configuration. Then the user can write a component that takes the helper as input and can use the helper's API to easily invoke their R script, which they store in their component's DLL as an embedded resource.

Each of our R-based statistics components follows a shared workflow. First, the component loads the experiment's sample data from the TraceLab workspace. Second, the component extracts and executes the R script corresponding to the statistics test in question. Finally, the component stores the resulting test statistic and p-value in the TraceLab workspace.

We unknowingly developed the ability to run R in parallel to similar efforts at the College of William and Mary. Their work, RPlugin, uses a similar technique to run R scripts, but uses a singleton pattern instead of providing a workspace variable. We only discovered this overlap inadvertently [67] and very late in development, so it should be interesting to compare the two implementations in future work. For now, we turn our attention to the main novelty, the MeansTest component.

9.5.2. **MeansTest Implementation**

Although R provides implementations for all of the tests mentioned in section III, R assumes that the user is a statistics savant who is aware of all of the assumptions that the tests entail. Our goal is to reduce the user's burden by cataloging and automatically testing these assumptions. TraceLab helped us in this respect by providing a very useful feature called *composite components*. The composite component wizard in TraceLab enables researchers to take a subset of an existing experiment and encapsulate it as one component. Using this wizard, we developed a composite component called MeansTest. The MeansTest component takes just a few parameters: the two samples the user is comparing, a flag specifying whether the samples represent paired data, and the Rscript opaque reference. After executing an experiment containing a MeansTest, TraceLab stores the p-value of the test and the appropriate test statistic in the

workspace. MeansTest then prints a human-readable summary of the steps and tests involved in the computation.

Figure 1 shows the TraceLab dependency graph of MeansTest.

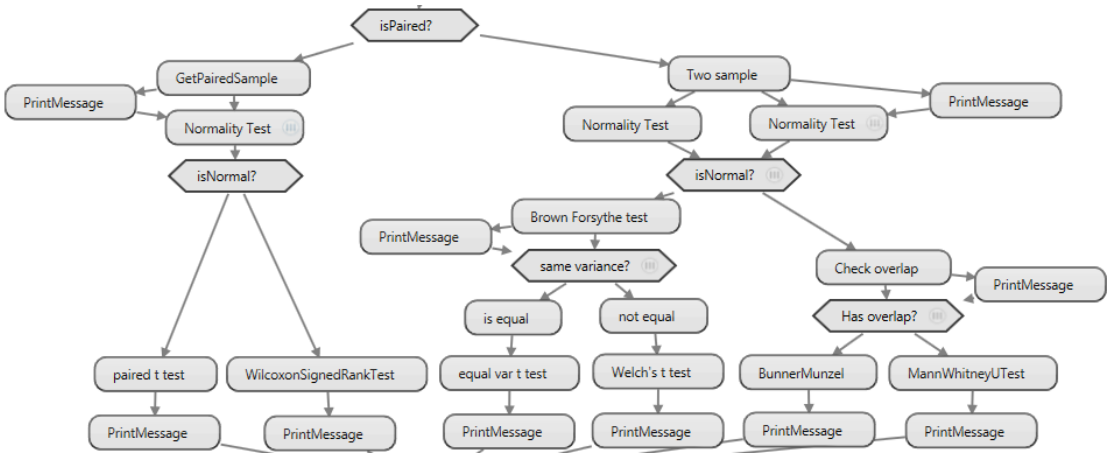

Figure 9.1. Internals of the MeansTest composite component.

As can be seen, MeansTest automatically verifies all of the assumptions one would normally have to check before performing a comparison of location parameters. First, MeansTest checks whether the user provided paired data. In the paired case, MeansTest branches to the left to test the normality of the pairs with Shapiro-Wilks. If the difference between the pairs is normally distributed, MeansTest performs the paired t-test. Otherwise, MeansTest performs the nonparametric Wilcoxon signed-rank test.

In the case the samples were not paired, MeansTest branches to the right and tests the normality of the two sample groups separately. If both groups are normal, MeansTest compares the sample variances for equality. If there is no evidence against this assumption, it performs a classical t-test using the pooled sample standard deviation; otherwise, it makes Welch's adjustment to the standard deviation when performing the t-test.

If either of the groups is not normal, MeansTest branches into the nonparametric tests. MeansTest checks the assumption of overlap between the samples. If this assumption holds, MeansTest simply invokes the Brunner-Munzel non-parametric test, which is designed for testing location differences in the presence of possibly unequal variances. When there is no overlap, then **clearly** the difference in means is significant, so MeansTest invokes the Mann Whitney U test only to provide the researcher with a non-zero p-value.

MeansTest is by no means a complete summary of all possible statistics tests, but is representative of the involved thought process we use in practice when comparing means. There are many other tests for normality, equal variance, and shift in location that fit specialized circumstances. There are also some assumptions, such as independence and identical distributions, which statisticians have yet to invent ways to numerically verify. We hope the TraceLab dependency graph of MeansTest will start a dialog in the statistics community to agree on a complete process for testing for shifts in means.

9.5.3. **Experiences**

In general, we found that the TraceLab tool was very stable and facilitated a wide variety of experiment procedures. As we mentioned earlier, composite components proved useful for bundling our massive statistics workflow into one comprehensive (and comprehensible!) statistics test. Besides applications in statistics, we found other uses for TraceLab. For instance, we were able to implement a classical mutation testing experiment comparing all-definitions testing to random testing. Mutation testing experiments are entirely outside the scope of TraceLab, yet the tool proved to be a plausible fit. Although the component framework adds extra work to experiment implementation, it is our experience that this extra work leads to portable experiments with reproducible results.

While the core tool provides a nice framework for developing experiments, we discovered several major issues indicating that the stock components are still experiencing growing pains. For instance, we identified a computation failure in the TraceLab vector space model where it reported that the cosine similarity between a vector and itself was far less than 1.0 [68]. This failure resulted from an error in the TF-IDF computation where the authors were normalizing the document vectors but not updating their pre-computed lengths. This mistake adversely affected the similarity measures; for example, Equation 1 gives the resulting erroneous cosine similarity:

$$\cos(q, d) = \frac{q \cdot d}{|q||d|^2} \quad (1)$$

The other measures were similarly impacted. For obvious reasons, this error invalidates the results of every tracing experiment run in TraceLab 0.5 or earlier using the default tracing components.

Errors like these aside, we see the need for design and documentation improvements to the stock components as well. For instance, it is not possible to extract the per-artifact recall and precision

scores from the experiment results data type; the data type hides these scores from the public API in the form of summary statistics. This API makes it impossible to perform meaningful analysis of experiment results. Also, the file importer and exporter descriptions provide no hint as to the expected file format. The worst offender is the "multiple dataset importer," which takes a "configuration file" as input. It would be useful to provide the expected formats in the components' descriptions.

Changes to the design of these components will definitely help improve productivity in TraceLab, but more work is needed. First, TraceLab needs to better advertise the availability of third-party components to collaborators. The Rscript/RPlugin overlap mentioned earlier is a perfect example of this necessity. We hope that the new Component Directory on coest.org [69] will help improve code reuse to avoid collisions like these.

Another key obstacle to productivity is that all operations, no matter how trivial, need to be encapsulated in their own components. For instance, to test the normality assumption in the paired case, one usually computes the difference between the two paired samples and tests the normality of the resulting vector. In R, this is very easy; if you have two vectors x and y, the expression x-y will return the input vector. However, TraceLab did not have a component to compute x-y, so we had to write our own x-y as a separate TraceLab component (see GetPairedSample in Figure 1) consisting of 99% TraceLab boilerplate and 1% actual code. We postulate that the existing decision nodes, which support inline scripts for making branching decisions, can be repurposed to avoid this boilerplate. To this end, we would like to see better documentation of decision nodes describing the available variables, the process to save workspace variables, and the particular .NET dialect in which decision code is written. Perhaps the TraceLab developers could help us create a facility to script R code inline as well.

9.6. Standard Language for Papers

"We used the statistical analysis components available in TraceLab. These were designed by computer scientists and statisticians at the University of Kentucky and use the well-respected statistical analysis toolkit R. The TraceLab components, collectively called MeansTest, first examine the paired or independent variables for the experiment and determine what statistical tests to apply by testing the appropriate assumptions. Next, the TraceLab components apply the appropriate statistical test. The components then report the appropriate p-value. This information

has been included below. Details on the statistical analysis methodology applied by the TraceLab component can be found in an earlier publication by Hays et al." (with proper citation of this TEFSE paper).

In addition, the researcher shall use the output from MeansTest to describe the tests applied. For example, "we had two samples, the data was normally distributed, we performed an F test and determined there was not equal variance in the samples, thus Welch's t-test was applied and yielded a p-value of NNN."

9.7. Evaluation

In order to vet the MeansTest components, we ran an independent evaluation. The dependent variable measures output by TraceLab from a typical traceability experiment on one dataset with comparison of techniques (collection of MAP values for a traceability dataset for TF-IDF with stopwords removed and MAP values for TF-IDF on that same dataset without stopword removal) was provided to the Statistics department co-authors of this paper. They independently analyzed the data using publicly available tools such as SAS (not R) and derived p-values (these are shown in Table 1). We generated t and p-values using MeansTest, also shown in Table 1. As expected, the values are within rounding error of each other. The t-distribution in this context is symmetric around zero, so the difference in sign simply reflects a minor implementation difference between their tools and R.

Table 9.1. Evaluation Results.

	Statistics Department values	**MeansTest values**
t	0.346225	-0.3462248
p-value	0.7371	0.7371301

9.8. Conclusions and Future Work

As the traceability research community ushers in the era of TraceLab, it will be much easier to generate and try out new ideas. It is incumbent upon researchers to practice responsible experimentation and use proper techniques in ensuring that the obtained results are statistically significant. Toward that end, we present the MeansTest component as well as standard language that can be used in papers which employ this composite TraceLab component. We evaluated our component and found that the values generated match those of SAS and similar tools.

In the future, we would like to expound on other statistical analyses such as power analysis and analysis of variance. As with the comparison of means, researchers performing other analyses of their results have many options readily available thanks to statistical software packages. Unfortunately, researchers often lack the required expertise to select the most appropriate option. While a fully automated solution to the selection process is not a panacea, we posit that such a solution can imitate the decisions of an expert in most applicable cases.

Chapter 10. Validation of Software Testing Experiments

10.1. Summary

Researchers in software testing are often faced with the following problem of empirical validation: does a new testing technique actually help analysts find more faults than some baseline method? Researchers evaluate their contribution using statistics to refute the null hypothesis that their technique is no better at finding faults than the state of the art. The decision as to which statistical methods are appropriate is best left to an expert statistician, but the reality is that software testing researchers often don't have this luxury. We developed an algorithm, MeansTest, to help automate some aspects of statistical analysis. We implemented MeansTest in the statistical software environment R, encouraging reuse and decreasing the need to write and test statistical analysis code. Our experiment showed that MeansTest has significantly higher F-measures than several other common hypothesis tests. We applied MeansTest to systematically validate the work presented at the 2013 IEEE Sixth International Conference on Software Testing, Verification, and Validation (ICST'13). We found six papers that potentially misstated the significance of their results. MeansTest provides a free and easy-to-use possibility for researchers to check whether their chosen statistical methods and the results obtained are plausible. It is available for download at coest.org.

10.2. Introduction

Research in software testing often lends itself to empirical validation; researchers show that some new way of discovering faults finds more faults or takes less time than the state of the art. Publications often describe such results as being *significant*. While significance is an overloaded term, in the context of empirical validation, we'd like to think that significance refers to *statistical significance*: that a difference between two means or variances is not likely due to chance. Significance in this context is demonstrated through formal hypothesis testing.

Researchers who ascribe to this notion of significance must also contend with threats to *statistical conclusion validity*. Researchers sometimes incorrectly fail to reject the null hypothesis due to choosing inappropriate statistical test. Worse, they almost never go back and analyze the *statistical power* of their experiment to determine whether their sample size was appropriate. Less often, they interpret a p-value as a probability when in fact the assumptions of the underlying statistical test have not been met. While the former problem can be solved by using powerful tests, powerful tests make assumptions that run the risk of suffering the latter problem. This impasse has led researchers in software testing to rely on classical, possibly rather conservative

tests from the Wilcoxon family because they "do not wish to make assumptions on the distribution." [70] We present relatively recent advances in nonparametric statistics that provide powerful alternatives to the classical tests.

In this paper, we make several contributions to address these issues. We introduce an algorithm, called MeansTest, to introduce the software testing research world to new statistical tests such as Brunner-Munzel. This process has the potential to automatically analyze experimental results. To our knowledge, there is no other approach readily available that goes through the workflow to analyze the data under study for properties such as normality, equal variance, and power, and then use that information to decide which test to apply, and then apply that test. We have encapsulated those aspects of a professional statistician's approach that can reasonably be automated; while it is obvious that common sense and experience of a trained statistician can't be replaced by an automated procedure, we also found that *some* crucial steps can indeed be left to software. The simulation of our algorithm demonstrates its usefulness. Nevertheless, the idea of automated test selection will always have an air of controversy and we recommend to the user to employ our automatism wisely, and with common sense. However, readers shall keep in mind that the only viable alternatives would be relying on unrealistic model assumptions, or restricting oneself to overly conservative tests and thus failing to detect important effects. We introduce a unique way to validate the statistical technique that blends statistics with classification-based validation seen in fields such as software fault classification. We use our statistical technique to examine prior art at the 2013 IEEE Sixth International Conference on Software Testing, Verification, and Validation (ICST'13) [71] for statistical significance.

This paper is organized as follows. Section III revisits the MeansTest implementation. Section IV discusses the challenges that researchers face in performing valid statistical analysis. In Sections V and VI, we analyze the effectiveness of MeansTest. Sections VII and IIX state the results of our meta-analysis of ICST'13. In Section IX, we discuss our planned future work.

10.3. MeansTest Algorithm

In our prior work [72], we introduced the first iteration of the MeansTest algorithm. We implemented it for the experiment design framework, TraceLab [73]. MeansTest automated important aspects of the basic logic that expert statisticians use when selecting statistics tests that compare location parameters, such as the mean and median. MeansTest implemented the

underlying statistical tests and testing of assumptions by invoking the statistics environment R. MeansTest reported the p-value, test statistic, and the logical path it took through the composite component. Our paper separately gave statistician-crafted language describing the meaning of each possible path so that researchers could correctly report the MeansTest output.

Figure 1 displays the precedence graph of the revised MeansTest algorithm used in this paper.

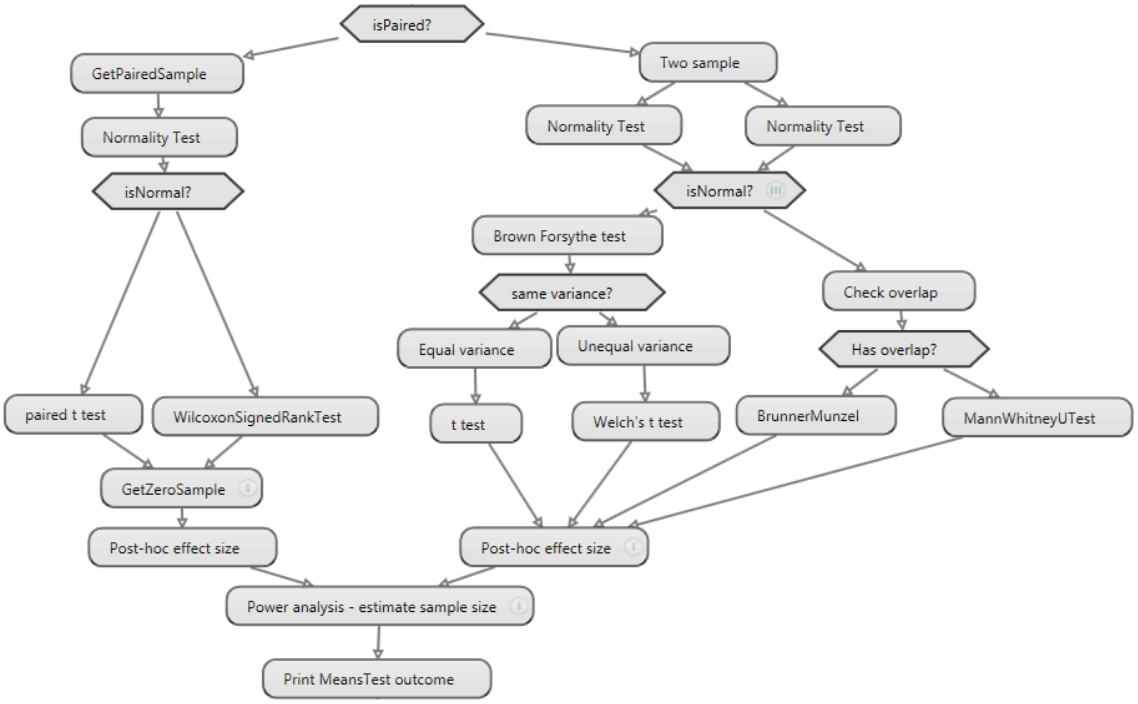

Figure 10.1. The MeansTest algorithm, as implemented in TraceLab.

MeansTest operates in two modes: paired-sample (aka "one-sample") and two-sample. In the paired mode, depending on the properties of the data being compared, MeansTest performs either the paired-sample t-test or Wilcoxon signed-rank test. In the two-sample mode, MeansTest performs one of: a) the t-test, b) Welch's t-test for unequal variances, c) Brunner-Munzel, or d) the Mann-Whitney U test (also known as the Wilcoxon rank-sum test), again depending on the shape and overlap of the two distributions.

Since its introduction, we have continued to improve to the MeansTest algorithm. We introduced post-hoc power analysis to facilitate pilot studies. In this context, power analysis takes experiment results that are not statistically significant and determines the minimum sample size required to make them significant; this analysis assumes that the observed results are not an

artifact of chance. Also, we now directly display in line with the results the descriptive text that a researcher can cite.

10.3.1. **Tests for Normality**

Razali and Wah [48] compared several algorithms for determining normality of data. Razali and Wah selected the four "most common" automated tests for normality: Shapiro-Wilk, Kolmogorov-Smirnov, Anderson-Darling, and Lilliefors. They applied these tests to samples of various sizes drawn from 14 probability distributions; they selected these distributions "to cover various standardized skewness and kurtosis values." They generated samples of sizes 10, 20, 30, 50, 100, 200, 300, 400, 500, 1000, and 2000. They found that the Shapiro-Wilk test was the most efficient overall at identifying non-normality.

The Razali and Wah paper is relevant to MeansTest for two reasons. First, they identified the best algorithm to determine non-normality; note that MeansTest used Shapiro-Wilk in the initial release. Second, they presented an empirical framework and data set that could be extended to validate MeansTest. Table 1 summarizes the probability distributions that Razali and Wah used. As Table 1 shows, the data is balanced evenly between symmetric and asymmetric distributions. It also has balanced skewness (seven values are 0) and kurtosis (seven absolute values are less than or equal to 1). This balance helps eliminate bias threats.

Table 10.1. Razali and Wah Distributions

Distribution	Skewness	Kurtosis
uniform(0,1)	0	-1.2
beta(2,1)	-0.5656854249	-0.6
beta(2,2)	0	-0.8571428571
beta(3,2)	-0.2857142857	-0.6428571429
beta(6,2)	-0.692820323	0.1090909091
t(7,0)	0	2
t(5,0)	0	6
t(10,0)	0	1
t(300,0)	0	0.0202702703
Laplace(0,1)	0	3
chisq(4,0)	1.4142135624	3
chisq(20,0)	0.632455532	0.6
Gamma(1,5)	2	6
Gamma(4,5)	1	1.5

10.4. The Importance of Statistical Analysis

Generally, statistical inference aims to quantify whether observed real-data phenomena could be explained by chance alone, or whether the observed data are so unusual that a convincing explanation requires a model with components beyond chance alone [74]. In the case of comparing two techniques, each technique results in a data sample. Most likely, the samples will differ from each other. However, even if both techniques are equivalent, one would expect the samples to be different due to chance variation. Statistical inference provides the tools to decide whether the two samples are different enough to reject the possibility that both methods were equally effective. To this end, statistical testing procedures yield p-values. A p-value in this context is the probability, assuming both methods are indeed equivalent, that two resulting data samples would be as (or more) different than the two samples that were observed in the experiment. P-values are one of the main decision tools in statistical inference. If the p-value is small, typically less than 0.05, researchers conclude that the assumption "both methods are indeed equivalent" can no longer be upheld.

One of the main problems with p-values is that the underlying probability calculation typically relies on several model assumptions. Unless these assumptions are carefully checked and verified, the seemingly precise "p-value" can be worthless and misleading. For example, the unpaired two-sample t-test can be used to compare two independent samples. Observations in each sample are assumed to be normally distributed with equal variance. If the samples are truly independent and the observations truly follow normal distributions and have the same variance, then a reported p-value of 0.03 can indeed be interpreted as a rejection of the hypothesis "both methods are equally effective," while a p-value of 0.12 does not provide evidence against this hypothesis. However, if the samples are actually paired instead of independent, this test will often provide large p-values even if both methods are rather different. The same can happen if the data is highly skewed and thus violates the normality assumption. On the other hand, it can also happen that an inappropriately chosen statistical test provides a small p-value even though the methods being compared are not distinguishable in quality, and the observed differences are in fact due to chance. Such a test is as undesirable as the first one. In either case, the resulting p-values do not serve as a meaningful decision tool, and they do not have the probability interpretation mentioned above.

The solution to this problem is rather straightforward: only appropriate inference procedures should be used. The MeansTest workflow facilitates this by making sure that all of the important

assumptions are being examined and that appropriate inference procedures are being chosen. As a result, the final reported p-value for the comparison of two methods still satisfies the probability interpretation given above. Also, this p-value can be used to decide whether the hypothesis "both techniques are equally effective" shall or shall not be rejected.

10.5. Experiment Design

In our experiment, we evaluated the accuracy of the MeansTest component. We drew samples at random from the Razali and Wah probability distributions. We drew second samples at varying distances from the original samples. We then applied hypothesis tests to see if they could notice the true difference in the population means.

10.5.1. **Research question**

As stated earlier, the MeansTest workflow combines several hypothesis tests that researchers already use. In light of the tendency of researchers to favor the Wilcoxon tests, we might ponder whether MeansTest is more effective overall than the Wilcoxon tests. We pose this hypothesis more generally in the form of RQ0 below.

RQ0: can MeansTest more accurately detect the true significance of differences/lack thereof than other hypothesis tests?

10.5.2. **Data**

We studied the Razali and Wah probability distributions from Table I.

10.5.3. **Procedure**

We expanded the Razali and Wah procedure to cover statistical significance. Razali and Wah only evaluated the likelihood that a test of normality could find a known difference in non-normality. We looked at two aspects: the likelihood that a hypothesis test would find a significant difference *when a difference was present*, and the likelihood that a hypothesis test would not find a significant difference *when no difference was present*. This procedure is more in line with the usual fault classification experiments in software testing and gives direct evidence toward answering our research question.

We applied the following hypothesis tests: MeansTest, Student's t-test, Welch's t-test, Wilcoxon ranked-sum, and Brunner-Munzel. As mentioned earlier, these tests are all invoked by MeansTest, so it is worth considering whether MeansTest can perform any better than its parts.

We ran our hypothesis tests on many pairs of samples. Each pair consisted of an initial sample and a shifted second sample. We drew our initial samples from the Razali and Wah distributions using their sample sizes. To draw the second samples, we used Cohen's effect size to define the difficulty of noticing a difference between two samples. We systematically examined 20 effect sizes: ten of the effect sizes were selected from zero to one in 0.1 increments, while the other ten had zero effect size. This selection created a balance between zero differences and non-zero differences, reducing bias. We selected the parameters of the second sample's probability distribution to yield the desired effect size. For each distribution/sample size/effect size triple, we drew 100 pairs of initial samples and shifted samples.

The result of running each hypothesis test on each pair of samples was a p-value. We interpreted the p-value at the 95% confidence level to determine whether an outcome was significant (positive) or not significant (negative). We interpreted the correctness of this result depending on the effect size. Table 2 concisely demonstrates our classification logic given a p-value *p* and an effect size *d*.

Table 10.2. Classification logic

	Positive	**Negative**
True	p<0.05,d>0	p>0.05,d=0
False	p<0.05,d=0	p>0.05,d>0

We labeled these quantities using *TP* for True Positive, *TN* for True Negative, *FP* for False Positive, and *FN* for False Negative. Using these labels, we then computed the F-measure of each classification. We use the following definitions to compute F-measure [75]:

$$\text{recall} = TP/(TP + FN) \tag{1}$$

$$\text{precision} = TP/(TP + FP) \tag{2}$$

$$F = 2 * \frac{\text{recall} * \text{precision}}{\text{recall} + \text{precision}}. \tag{3}$$

The F-measure of all hypothesis tests increased with sample size, but the relative rankings between methods remained stable. To eliminate the effect of sample size on F-measure, we ordinally ranked each value at each sample size; the ranks were consistent across sample sizes. This consistency allowed us to summarize our results by distribution.

10.5.4. **Hypothesis**

We wanted to show that MeansTest had **higher rank** than that of other hypothesis tests. Given a hypothesis test *i*, Equations 4 and 5 formally state the null and alternate hypotheses as:

$$H_0: \text{Rank}(\text{MeansTest}) = \text{Rank}(i) \quad (4)$$

$$H_A: \text{Rank}(\text{MeansTest}) > \text{Rank}(i). \quad (5)$$

We rejected each null hypothesis with 95% confidence.

10.6. Results

In this section, we describe our results, including the rankings by distribution, the p-values for the hypothesis tests, and the threats to validity.

10.6.1. **Rankings**

Table 3 shows the summary of the ranks on an ordinal scale of 1-5, using the average for ties. **Higher** ranks are better.

Table 10.3. Hypothesis Test Rankings

Distribution	Means-Test	Wilcoxon	t	Welch t	Brunner-Munzel
beta(2,1)	5	3	1.5	1.5	4
beta(2,2)	3	1	4	5	2
beta(3,2)	3	1	5	4	2
beta(6,2)	5	3	2	1	4
chisq(20,0)	4	3	2	1	5
chisq(4,0)	3.5	3.5	2	1	5
Gamma(1, 5)	4.5	3	2	1	4.5
Gamma(4, 5)	5	2	3	1	4
Laplace(0, 1)	3	5	2	1	4
t(10,0)	4	3	2	1	5
t(300,0)	4	1	5	3	2
t(5,0)	3	4	2	1	5
t(7,0)	3	4	2	1	5
uniform(0, 1)	3	1	5	4	2

As Table 3 shows, MeansTest demonstrated favorable classification behavior over the other tests. While the individual tests each had their strong and weak points, MeansTest was able to infer the

appropriate test sufficiently to **always place at least third or higher**. Contrast that with the Wilcoxon rank-sum test and Welch's t-test, which both fared poorly with alarming frequency. Brunner-Munzel, the new non-parametric test, usually did well. With regard to the added entry on normality, the t-tests nicely complemented Brunner-Munzel's weak points, lending credibility to our idea to pick between the tests based on the normality of the data.

10.6.2. **Summary statistics**

Table 4, in turn, provides summary statistics for Table 3.

Table 10.4. Hypothesis Test Summary Statistics

Test	Worst rank	Best rank	Mean	p-value
MeansTest	**3**	5	3.8	-
Wilcoxon	1	5	**2.7**	**0.013**
Welch's t	1	5	**1.9**	**0.0037**
t	1.5	5	**2.8**	**0.038**
Brunner-Munzel	2	5	3.8	0.61

To evaluate our set of hypotheses regarding the ranks, we applied paired hypothesis testing to account for the fact that the ranks are dependent variables on the distribution being tested. Since we are here comparing ranks, only a nonparametric rank test is appropriate. Note that ranks have the property that their sum is always constant (consider the row sums in Table 3) – therefore violating an always implicitly assumed independence assumption in parametric tests.

Using this information, we applied the Wilcoxon signed-rank test (as Brunner-Munzel only applies to independent samples). As Table 4 highlights, we found that MeansTest had a **significantly higher** rank than Wilcoxon rank-sum and the t-tests. MeansTest did not have a significantly higher rank than Brunner-Munzel.

10.6.3. **Threats to Validity**

In terms of threats to *statistical conclusion validity*, our results only have 95% confidence. We declared our hypotheses ahead of time so we would not have to worry about inflated experiment-wide error from performing multiple comparisons. Even if we had decided on all four tests after performing the experiment, we should still have at least 80% experiment-wide confidence according to the very conservative Bonferroni correction.

In terms of threats to *construct validity*, these should be minimal because we used the statistics framework R to perform all of our statistics. In our previous work [72], we tested each MeansTest

path and confirmed that it returned the same result as the expert statisticians using other statistics software. Similarly, in our experiment, we used R to generate the probability distributions and shift their parameters.

In terms of threats to *internal validity*, we used a data set that was previously used in an experiment to assess the ability of statistics to infer non-normality, which on the surface could appear to create a bias towards nonparametric statistics. We defused this threat by establishing that the distributions were balanced between normal and non-normal skewness and kurtosis. Many of the distributions used in the experiment, such as the t distribution, were in fact approximately normal. We ran each hypothesis test on every data set we generated, so it is not possible that MeansTest received "easier" samples than the other tests.

In terms of threats to *external validity*, we only looked at the 14 Razali and Wah distributions. While we generated many samples from these distributions, it is true that there are other distributions out there such as the negative exponential and standard normal distributions. This threat is mitigated by Razali and Wah's methodology, in that they selected distributions to cover a range of standard skewness and kurtosis values. Thus, the skewness and kurtosis of many other distributions are implicitly covered by these 14 distributions.

10.7. Meta-analysis

Borrowing from the Dit et al. mapping process, we systematically mapped the proceedings of ICST'13 into our experiment framework. We applied MeansTest to the experiments from those papers that 1) featured empirical comparisons of two or more testing methods/tools, and 2) had sufficient data in the paper to perform the validation. Based on the output from MeansTest, we reported the statistical significance of the experiments' results and provided recommendations for insignificant results.

10.7.1. **Research Questions**

We are curious to know:

RQ1: How pervasive were Wilcoxon tests at ICST'13?

RQ2: To what extent are the results at ICST'13 statistically significant?

10.7.2. **Conducting the search**

We systematically examined prior art from ICST'13. As we will show, ICST'13 was of interest because the venue featured considerable empirical validation using the Wilcoxon family of tests. ICST'13 also had several empirical validation papers which did not comment on the statistical significance of their results.

10.7.3. **Screening criteria**

We wanted to analyze the existing published results of papers at ICST'13, so we *included* papers which consisted of empirical studies of testing methods that published their data. We *excluded* other types of papers such as practical experience reports and papers with formal proofs.

10.7.4. **Classification**

We classified papers at several levels: their track, their focus, whether they published raw data, and whether the results were statistically significant. We considered a result statistically significant if MeansTest reported at least one significant result and MeansTest found at least as many significant results as the authors claimed; if MeansTest disagreed with the authors about the significance of their results, we classified that paper overall as not being statistically significant. In this way, we did not bias our classification against thorough experiments with many hypothesis tests and some statistically insignificant results.

10.7.5. **Data extraction**

There were four pieces of data we extracted from each paper: 1) the paper's hypotheses, 2) the hypothesis testing applied (if any), 3) the published results and claims of significance, and 4) the MeansTest assessment of the results. We extracted these through manual reading and copy/paste of tables. Whereas Dit et al. reproduced entire experiments and published the TraceLab components implementing them, we found that we could sufficiently answer our more modest research questions through meta-analysis of the published results.

Figure 2 shows the workflow we created to model individual comparisons of means in software testing experiments. Based on the paper's hypotheses, we manually established the nature of the samples (paired *vs.* two independent samples) as an input to the overall workflow. After inputting each sample, we executed the MeansTest workflow depicted in Figure 1, abstracted here as the node labeled MeansTest. As an output, MeansTest provided information such as the p-value, the hypothesis test used, and the sample size required to get a significant result. We compared those results with the results the authors provided. We used this workflow to present the results below.

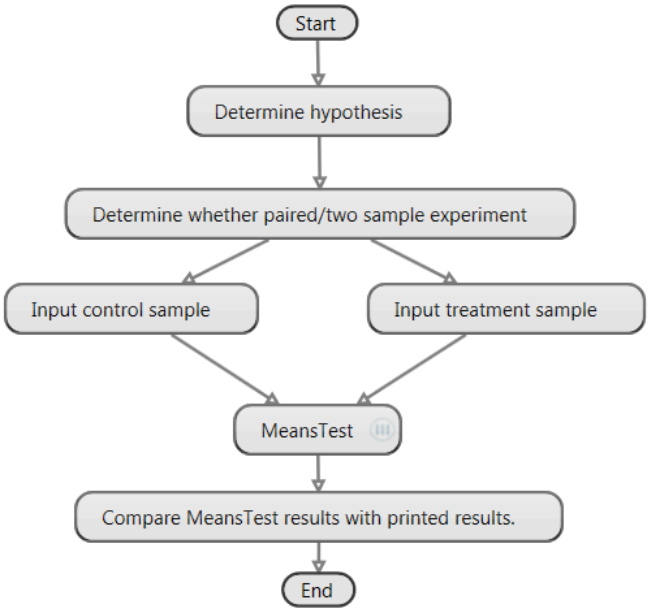

Figure 10.2. Workflow for ICST 2013 meta-analysis.

10.8. Meta-analysis results

Figure 3 shows the scope of the meta-analysis.

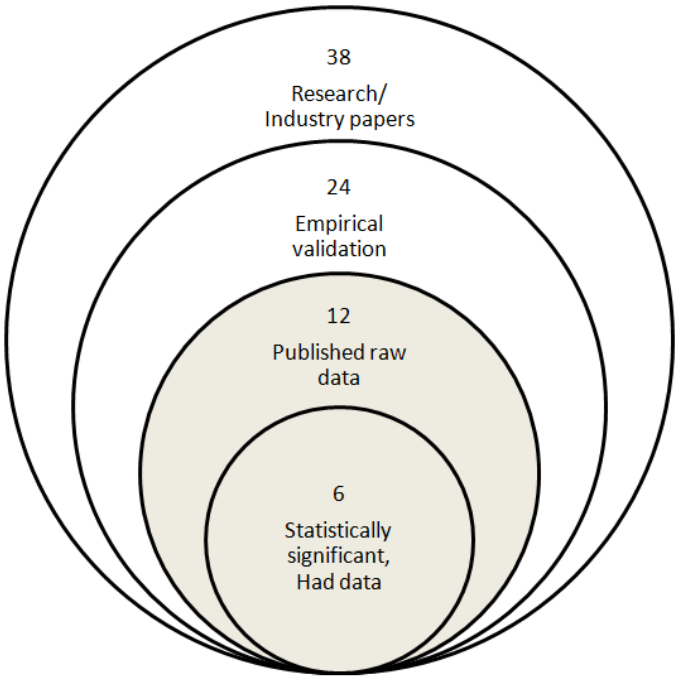

Figure 10.3. Scope of ICST 2013 meta-analysis.

There were **38** papers in the main testing and industry track. Of those, **24** papers (63%) featured some kind of empirical validation comparing a method with one or more baseline methods. Of those 24 papers, only eight papers (33%) reported the statistical significance of the authors' empirical validation. In response to **RQ1**, we note that **six papers (75%) used the Wilcoxon tests** with no consideration for alternative tests. Only one of these papers [76] printed enough raw data to enable a meta-analysis; the remaining 7 papers reported only summary statistics that we could not validate with MeansTest.

We examined the remaining 16 papers with no statistical validation for the presence of raw data. Of the 16 papers, 11 papers (69%) printed enough of their experiment data in the proceedings to suffice for a MeansTest meta-analysis. Together with the Canfora et al. paper, we analyzed **12** papers with data.

In response to **RQ2**, our meta-analysis found that **only six papers (50%)** had reproducible statistically significant results at the 95% confidence level. This proportion is consistent with the Kampenes et al. systematic review [45]. The remaining six papers invariably claimed "significant" results even though the experiment was too small to support the *statistical* significance of said results. These issues in the non-significant experiments could likely be remedied with a larger experiment. In the next sections, we use the MeansTest power analysis to recommend appropriate sample sizes.

10.8.1. **Analyses by experiment**

In this section, we briefly summarize the 12 experiments in question. We report the properties of the experiment as inferred by MeansTest, including normality, the appropriate hypothesis test, and MeansTest's p-value for the experiment. In cases where the authors' results were not statistically significant, we also state MeansTest's power analysis to suggest the appropriate course of action in order to get a significant result.

Multi-Objective Cross-Project Defect Prediction

Canfora et al. [76] introduce a regression model, which they call Multiple-Objective Logistic Regression, to predict defects across projects. They compare their model with a Within-Project Logistic model, a Single-Objective cross-project Logistic model, and a Clustering-Based Logistic model. They study 10 projects. For each project, they compute the cost of the model, its recall, and precision.

They concede that the Within-Project model **is better** than their cross-project model. **Using the Wilcoxon signed-rank test**, they report that the Multiple-Objective Logistic model **significantly diverges** from the Single-Objective Logistic model in terms of the cost (p=0.02), but not the precision (p=0.4). Finally, they report that the Multiple-Objective Logistic model **significantly diverges** from the Clustering-Based Logistic model in terms of the cost (p=0.009) but the precision is **borderline significant** (p=0.05). All of the models had identical recall.

This paper is of particular interest to this meta-analysis because it features raw data, existing statistical analysis of the results, and a statistically borderline p-value of 0.05. Better still, this inconclusive p-value was achieved because the authors used the least-powerful Wilcoxon test without justification. MeansTest was designed to address exactly this situation by automatically inferring whether the data is normal to lend more power to the analysis when appropriate.

Table V summarizes our meta-analysis of this paper. Most of our results were the same, but in the case of the borderline significant p-value, MeansTest concluded that the data was sufficiently normally distributed to apply the t-test; this enabled the difference in precision to **become statistically significant**.

Table 10.5. Canfora et al. vs MeansTest

Logistic model	**Cost p-value**		**Precision p-value**	
	Author's	***MeansTest***	***Author's***	***MeansTest***
Within-Project	-	0.15	-	0.06
Single-Objective	0.02	0.02	0.4	1.0
Clustering-Based	0.009	0.01	**0.05**	**0.02**

In light of this small victory, one might pause to ponder whether the MeansTest p-values, *themselves*, are significantly different from the p-values of Canfora et al. and the broader research community. One could recursively apply MeansTest to the MeansTest p-values and the authors' p-values to make that determination. If the difference were not significant, the MeansTest power analysis would recommend the required sample size needed to get a significant difference. We leave this problem as future work.

Empirical Evaluation of the Statement Deletion Mutation Operator

Deng et al. [77] examined the effectiveness of the statement deletion mutation operator. They applied this mutation to 40 Java classes. For each class, the first author built a test set that killed every deletion mutant. They then applied the test set to muJava's mutants. They reported that the deletion operator used significantly **less** mutants than muJava to get **roughly the same** level of coverage as a test set generated from killing muJava mutants.

In applying MeansTest, we found that neither of the paired data sets were normally distributed. The statement deletion operator indeed produced **significantly less** mutants than muJava (Wilcoxon signed-rank, p~10^-8). However, MeansTest reported that the deletion operator had a **significantly worse** mutation score than a muJava test set with mutation score 1 (Wilcoxon signed-rank, p ~10^-7).

Symbolic Path-Oriented Test Data Generation for Floating-Point Programs

Bagnara et al. [78] introduced a performance optimization to the symbolic constraint solver for C code, FPSE. They ran their improved code against the stock code and measured the running time against 1-12 iterations of the C functions `dichotomic()` and `tcas_periodic_task_1Hz()`. They reported **improved execution times**, including solving some problems that caused the original code to time out.

MeansTest inferred that the performance data was approximately normal under `dichotomic()`, but not under `tcas_periodic_task_1Hz()`. The performance optimization was indeed **significantly faster** under `dichotomic()` (t-test, p=0.02) but **not significantly faster** under `tcas_periodic_task_1Hz()` (Wilcoxon signed-rank, p-value=0.12). According to MeansTest's power analysis, the authors would need to run at least **55 iterations** to get a statistically significant difference in performance.

Generating Effective Integration Test Cases from Unit Ones

Pezzè et al. [79] developed an Eclipse plugin, called Fusion, for automatically generating integration test cases from the semantics of unit test cases. They compared the number of faults and false positives found by their method with two other tools: Randoop and Palus. They performed this comparison across four programs. They reported that Fusion found **different faults** than Randoop and Palus, but had a **comparable number** of false positives.

We applied MeansTest 4 times total to compare Fusion with the other two methods. MeansTest inferred that the differences between the methods were normally distributed. It is difficult to

formulate a hypothesis for assessing the statistical significance of finding "different" faults, but Fusion **did not find** significantly different number of real faults than either Randoop or Palus (t-test, p=0.09 and 0.13, respectively). Fusion indeed found about the same false positives as Randoop and Palus (t-test, p=0.26 and 0.12, respectively). According to MeansTest's power analysis, the authors would need to test at least **12 programs** to notice a difference in real faults, and **26 programs** to notice a difference in false positives.

Improving Test Generation under Rich Contracts by

Tight Bounds and Incremental SAT Solving

Abad et al. [80] developed a new test generator, called FAJITA. They compared the branch coverage and performance of FAJITA with Pex, Kiasan, Randoop, AutoTest, and EvoSuite. They ran these tools on 25 methods across 8 classes. They reported that FAJITA **had the best branch coverage** of all tools.

We applied MeansTest 4 times to compare FAJITA's branch coverage to that of each of the other tools. We configured MeansTest to state the authors' hypothesis as one-sided. MeansTest inferred that the differences between the tools were not normally distributed. FAJITA did have **significantly greater** branch coverage than Pex, Kiasan, Randoop, AutoTest, and EvoSuite (Wilcoxon signed-rank, p=0.0008, 0.03, 0.0002, 0.0004, 0.01, respectively).

Search-Based Testing of Relational Schema Integrity Constraints Across Multiple Database Management Systems

Kapfhammer et al. [81] developed an input generator, called AVM, to test the constraints on database schemas. They compared the constraint coverage of AVM to DBMonster. They reported that AVM had **better constraint coverage** than DBMonster.

MeansTest inferred that the difference between the data was not normally distributed. MeansTest concluded that the constraint coverage of AVM was **significantly better** than DBMonster (Wilcoxon, p~10^-5).

MFL: Method-Level Fault Localization with Causal Inference

Shu et al. [82] applied spectrum-based fault localization at the method level. They compared their technique, MFL, to existing SBFL tools Tarantula, Ochiai, PFIC, and one based on the F-measure. They ran these tools across 4 programs each seeded with about 7 faults and calculated the minimum cost of a developer searching through methods according to the suspiciousness

ranks to find a bug. The authors reported that MFL **was cheaper** to use than the other measures in 3 out of 4 programs.

MeansTest inferred that the difference between the minimum costs of the methods was normally distributed. MeansTest concluded that the minimum cost of MFL was **significantly cheaper** than Tarantula and PFIC (t-test, p=0.033 and 0.34, respectively), but **not significantly** cheaper than Ochiai and F-measure (t-test, p=0.07 and 0.06, respectively). According to MeansTest's power analysis, the authors would need to test at least **11 programs** to notice a difference in minimum cost in all four tools.

Scaling Model Checking for Test Generation using Dynamic Inference

Yeolekar et al. [83] developed a test generation tool, called AutoGen, for satisfying structural coverage criteria. They compared the branch coverage of test cases generated with their tool against random testing and another test generator called SatAbs. They applied these tools to 10 functions. They reported that AutoGen had **better coverage** than SatAbs and random testing.

AutoGen had **significantly higher** coverage than random (t-test, p~10^-5). The analysis of the difference between SatAbs and AutoGen is tricky because although SatAbs had higher coverage in some instances, it timed out on most functions. If we treat the timeouts as 0% coverage, we see that AutoGen had **significantly higher** coverage than SatAbs, but the difference was not normally distributed (Wilcoxon signed-rank test, p= 0.04).

Transformation Rules for Platform Independent Testing: An Empirical Study

Eriksson et al. [84] introduced UML transformations to identify implicit logical predicates ahead of time, before code is generated from the models. Their goal was to reduce the number of requirements needed to satisfy logic coverage criteria such as all-pairs and MCDC. They examine the UML of 6 programs and apply their transformations to the programs. They then compute the number of new requirements generated going from UML to code and show that their method requires **less** new rules.

MeansTest inferred that the data was not normally distributed. The implicit-to-explicit transformations did indeed generate **significantly less** rules going from the UML to code; this result applied to both all-pairs and MCDC coverage requirements (Wilcoxon, p=0.008).

An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation

Yu et al. [85] introduced their combinatorial test generation tool, called ACTS. They compared ACTS to other combinatorial test generators: CASA, Ttuples, and PICT. They compared the tools' performance in terms of the amount of time spent building the test set. They evaluated their tools across 16 programs and concluded that "ACTS can perform significantly **better** for systems with more complex constraints."

MeansTest inferred that the performance data was not normally distributed. ACTS was **significantly faster** than CASA and TTuples, but **not** PICT (Wilcoxon, p=0.0001, 0.004, and 0.37, respectively). MeansTest estimates that the authors would need **35 programs** to show a statistically significant difference in the runtime performance between ACTS and PICT.

Oracle-Based Regression Test Selection

Yu et al. [86] examined the problem of regression test selection as part of change impact analysis at ABB. They discussed two broad methods of creating test oracles: using outputs and tracking the internal state. They introduced an algorithm for inferring the test cases needed to test a change, based on so-called "internal oracles" that study the effect of changes on the internals of a system. They compared the faults found by test sets selected by internal oracles with those generated by "output oracles" (oracles that only check the output) on 9 programs. They found that internal oracles discover **significantly more** faults than output oracles.

MeansTest inferred that the fault distribution data was normally distributed. The internal oracle tests found **significantly more** faults than the output oracle tests (t-test, p=0.002).

Test Case Prioritization Using Requirements-Based Clustering

Arafeen and Do [87] examined the issue of test case prioritization: which test cases are most likely to uncover faults? They introduced a new test case clustering technique that orders test cases based on the priority of their requirements. They introduce several within-cluster ordering heuristics as well. They compare the effectiveness of prioritizing with clustering against standard McCabe-style prioritization metrics on four programs: three versions of iTrust and Capstone. They find that clustering **outperforms** McCabe on iTrust, but **not** on Capstone.

MeansTest inferred that the relative effectiveness percentages were normally distributed. The clustering technique **significantly outperformed** McCabe on the iTrust code (t-test, p~10^-14) but **not** on Capstone (t-test, p=0.18). The Capstone program was too small to perform a meaningful analysis.

10.9. Discussion and Future Work

An automated selection process for statistical analysis can help researchers draw conclusions about the statistical significance of their results in the absence of an expert statistician. Our validation showed that this process significantly outperformed blind adherence to the Wilcoxon nonparametric tests. In light of newer nonparametric hypothesis tests such as Brunner-Munzel, the adherence to the Wilcoxon tests that prevailed in ICST'13 may be outdated. Indeed, we found a specific instance at ICST'13 where our process found a significant result that was originally reported as being of questionable significance. While very promising, our results indicate that our process is still not perfect; it is not a substitute for an expert statistician. It is ultimately up to an expert to decide which test is most appropriate in a given situation.

Cross-referencing our meta-analysis with the systematic review by Kampenes et al. [45], we see that ICST'13 had almost identical statistical significance as journal papers. About 50% of results were statistically significant in both studies. Unfortunately, authors at ICST'13 under-reported the statistical significance of their work compared to the Kampenes et al. journal papers, with only 33% of empirical validation papers at ICST'13 reporting their statistics. We hope that our workflow will make it more convenient for authors in the future to report statistical significance.

In our meta-analysis of ICST'13, we stated the minimum sample sizes required to achieve statistically significant results. Our methodology, well-known to statisticians as power analysis, is a welcome addition to the automated statistics mode of thought. Many problem domains call for even more sophisticated analysis, such as blocked designs and analysis of variance, for which we have yet to provide a solution. There are several models of analysis of variance, each with their own assumptions, so this type of analysis would benefit from an automated selection process similar to that of MeansTest and remains as future work.

References

[1] "Free NIST Software Tool Boosts Detection of Software Bugs | Department of Commerce." [Online]. Available: http://www.commerce.gov/blog/2010/11/09/free-nist-software-tool-boosts-detection-software-bugs. [Accessed: 10-Feb-2014].

[2] "Issa: HealthCare.gov costs could top $1 billion | TheHill." [Online]. Available: http://thehill.com/blogs/healthwatch/health-reform-implementation/192099-issa-healthcaregov-costs-could-surpass-1. [Accessed: 10-Feb-2014].

[3] A. Abran and J. W. Moore, *Guide to the software engineering body of knowledge: trial version*. Los Alamitos, Calif.: IEEE Computer Society, 2001.

[4] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," 2011, p. 1.

[5] "IEEE Standard for Software and System Test Documentation," IEEE Standard 829-2008, 2008.

[6] "IEEE Standard Glossary of Software Engineering Terminology," IEEE Standard 610.12-1990, 1990.

[7] "IEEE Standard for Property Specification Language (PSL)," IEEE Standard 1850-2005, 2005.

[8] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge: Cambridge University Press, 2008.

[9] "IEEE Standard Dictionary of Measures of the Software Aspects of Dependability," IEEE Standard 982.1-2005, 2006.

[10] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria," in *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, Victoria, British Columbia, Canada, 1991, pp. 154–164.

[11] B. Dit, E. Moritz, M. Linares-Vasquez, and D. Poshyvank, "Supporting and Accelerating Reproducible Research in Software Maintenance using TraceLab Component Library," in *Proceedings of 29th IEEE International Conference on Software Maintenance (ICSM'13)*, Eindhoven, the Nethelands, 2013.

[12] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *Proceedings of the 12th international conference on Evaluation and Assessment in Software Engineering*, Swinton, UK, UK, 2008, pp. 68–77.

[13] V. Basili, R. Selby, and D. Hutchens, "Experimentation in Software Engineering," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 7, pp. 733–743, Jul. 1986.

[14] A. J. Offutt and J. H. Hayes, "A semantic model of program faults," *ACM SIGSOFT Softw. Eng. Notes*, vol. 21, no. 3, pp. 195–200, May 1996.

[15] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti, "XACMUT: XACML 2.0 Mutants Generator," 2013, pp. 28–33.

[16] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," 2007, p. 667.

[17] T. Mouelhiv, F. Fleurey, and B. Baudry, "A Generic Metamodel For Security Policies Mutation," 2008, pp. 278–286.

[18] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, pp. 5–20, Jan. 1992.

[19] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," 2007, pp. 89–98.

[20] M. H. Halstead, *Elements of software science*. New York: Elsevier, 1977.

[21] P. Bellini, I. Bruno, P. Nesi, and D. Rogai, “Comparing Fault-Proneness Estimation Models,” pp. 205–214.
[22] F. Lanubile, A. Lonigro, and G. Visaggio, “Comparing Models for Identifying Fault-Prone Software Components,” presented at the 7th International Conference on Software Engineering and Knowledge Engineering, 1995, pp. 312–319.
[23] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, “Where the bugs are,” 2004, p. 86.
[24] H. Hata, O. Mizuno, and T. Kikuno, “An extension of fault-prone filtering using precise training and a dynamic threshold,” in *Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR)*, Leipzig, Germany, 2008, pp. 89–98.
[25] T. J. McCabe, “A Complexity Measure,” *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.
[26] M. Hays and J. Hayes, “The Effect of Testability on Fault Proneness: A Case Study of the Apache HTTP Server,” in *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, 2012, pp. 153–158.
[27] J. M. Voas, “PIE: a dynamic failure-based technique,” *IEEE Trans. Softw. Eng.*, vol. 18, pp. 717–727, Aug. 1992.
[28] J. M. Voas and K. W. Miller, “Software testability: the new verification,” *IEEE Softw.*, vol. 12, no. 3, pp. 17–28, May 1995.
[29] R. S. Freedman, “Testability of software components,” *IEEE Trans. Softw. Eng.*, vol. 17, no. 6, pp. 553–564, Jun. 1991.
[30] R. Santelices and M. J. Harrold, “Probabilistic Slicing for Predictive Impact Analysis,” GIT-CERCS-10-10.
[31] N. Li, F. Li, and J. Offutt, “Better Algorithms to Minimize the Cost of Test Paths,” in *Fifth International Conference on Software Testing, Verification and Validation (ICST)*, Montreal Canada, 2012.
[32] A. Filieri, C. S. P\uas\uareanu, and W. Visser, “Reliability Analysis in Symbolic Pathfinder,” in *Proceedings of the 2013 International Conference on Software Engineering*, Piscataway, NJ, USA, 2013, pp. 622–631.
[33] “Java Path Finder.” [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf. [Accessed: 22-Jan-2012].
[34] “How do I compute the R-square statistic for ROBUSTFIT using Statistics Toolbox 7.0 (R2008b)? - MATLAB & Simulink.” [Online]. Available: http://www.mathworks.com/support/solutions/en/data/1-CMABGO/index.html?product=ST&solution=1-CMABGO. [Accessed: 07-Feb-2012].
[35] J. Hayes, A. Dekhtyar, and J. Osborne, “Improving Requirements Tracing via Information Retrieval,” in *Proceedings of the International Conference on Requirements Engineering (RE)*, 2003, pp. 151–161.
[36] S. Henry and D. Kafura, “Software Structure Metrics Based on Information Flow,” *IEEE Trans. Softw. Eng.*, vol. SE-7, no. 5, pp. 510–518, Sep. 1981.
[37] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Hillsdale, N.J: L. Erlbaum Associates, 1988.
[38] Student, “The Probable Error of a Mean,” *Biometrika*, vol. 6, no. 1, p. 1, Mar. 1908.
[39] F. Wilcoxon, “Individual Comparisons by Ranking Methods,” *Biom. Bull.*, vol. 1, no. 6, p. 80, Dec. 1945.
[40] E. Brunner and U. Munzel, “The Nonparametric Behrens-Fisher Problem: Asymptotic Theory and a Small-Sample Approximation,” *Biom. J.*, vol. 42, no. 1, pp. 17–25, Jan. 2000.
[41] K. Pearson, “‘ Das Fehlergesetz und Seine Verallgemeinerungen Durch Fechner und Pearson.’ A Rejoinder,” *Biometrika*, vol. 4, no. 1/2, pp. 169–212, 1905.

[42] S. S. Shapiro and M. B. Wilk, “An analysis of variance test for normality (complete samples),” *Biometrika*, vol. 52, no. 3–4, pp. 591–611, Dec. 1965.
[43] M. B. Brown and A. B. Forsythe, “Robust Tests for the Equality of Variances,” *J. Am. Stat. Assoc.*, vol. 69, no. 346, pp. 364–367, Jun. 1974.
[44] R Development Core Team, “R: A Language and Environment for Statistical Computing,” 2008. [Online]. Available: http://www.R-project.org.
[45] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg, “A systematic review of effect size in software engineering experiments,” *Inf. Softw. Technol.*, vol. 49, no. 11–12, pp. 1073 – 1086, 2007.
[46] B. L. Welch, “The significance of the difference between two means when the population variances are unequal,” *Biometrika*, vol. 29, no. 3–4, pp. 350–362, Feb. 1938.
[47] P. Breheny, “Relative e_fficiency,” presented at the STA 621: Nonparametric Statistics, University of Kentucky.
[48] N. M. Razali and Y. B. Wah, “Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests,” *J. Stat. Model. Anal. Vol*, vol. 2, no. 1, pp. 21–33, 2011.
[49] G. Denaro, “Estimating software fault-proneness for tuning testing activities,” in *Proceedings of the 22nd international conference on Software engineering*, Limerick, Ireland, 2000, pp. 704–706.
[50] “10 Dollars Bug Fix on Dilbert.com.” [Online]. Available: http://search.dilbert.com/comic/10%20Dollars%20Bug%20Fix. [Accessed: 10-Sep-2012].
[51] “Home | Graphviz - Graph Visualization Software.” [Online]. Available: http://www.graphviz.org/. [Accessed: 29-Mar-2011].
[52] “GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF).” [Online]. Available: http://gcc.gnu.org/. [Accessed: 10-Sep-2012].
[53] “SSA for Trees - GNU Project - Free Software Foundation (FSF).” [Online]. Available: http://gcc.gnu.org/projects/tree-ssa/. [Accessed: 10-Sep-2012].
[54] “The GNU Awk User’s Guide.” [Online]. Available: http://www.gnu.org/software/gawk/manual/gawk.html. [Accessed: 10-Sep-2012].
[55] “gcc(1): GNU project C/C++ compiler - Linux man page.” [Online]. Available: http://linux.die.net/man/1/gcc. [Accessed: 10-Sep-2012].
[56] Y.-S. Ma, J. Offutt, and Y. R. Kwon, “MuJava: an automated class mutation system,” *Softw. Test. Verification Reliab.*, vol. 15, no. 2, pp. 97–133, Jun. 2005.
[57] “SourceForge.net: MuJava Users - guitar.” [Online]. Available: http://sourceforge.net/apps/mediawiki/guitar/index.php?title=MuJava_Users. [Accessed: 19-Jan-2014].
[58] J. Offutt, *Trityp*. 2003.
[59] “Diffutils - GNU Project - Free Software Foundation.” [Online]. Available: http://www.gnu.org/software/diffutils/. [Accessed: 19-Jan-2014].
[60] W.-K. Kong, J. H. Hayes, A. Dekhtyar, and O. Dekhtyar, “Process improvement for traceability: A study of human fallibility,” presented at the Requirements Engineering Conference (RE), 2012 20th IEEE International, 2012, pp. 31–40.
[61] N. Niu and A. Mahmoud, “Enhancing candidate link generation for requirements tracing: The cluster hypothesis revisited,” presented at the Requirements Engineering Conference (RE), 2012 20th IEEE International, 2012, pp. 81–90.
[62] Y. Shin, J. Hayes, and J. Cleland-Huang, “A Framework for Evaluating Traceability,” DePaul University Technical Report TR12-001.

[63] G Deuchler, “Über die Methoden der Korrelationsrechnung in der Pädagogik und Psychologie,” *Ztg. Für Pädagog. Psychol.*, no. 15, pp. 114–131, 145–159, 229–242, 1914.
[64] H. B. Mann and D. R. Whitney, “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other,” *Ann. Math. Stat.*, vol. 18, no. 1, pp. 50–60, Mar. 1947.
[65] H. Levene, “Robust tests for equality of variances,” *Contrib. Probab. Stat. Essays Honor Harold Hotell.*, pp. 278–292, 1960.
[66] “ALGLIB, Cross Platform Numerical Analysis Library.” [Online]. Available: http://www.alglib.net/. [Accessed: 26-Mar-2013].
[67] B. Dit, E. Moritz, and D. Poshyvanyk, “A TraceLab-based solution for creating, conducting, and sharing feature location experiments,” in *Proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC)*, 2012, pp. 203–208.
[68] “Bug #200: VSM returns strange values - TraceLab - Projects.” [Online]. Available: http://coest.org/coest-projects/issues/200. [Accessed: 12-Feb-2013].
[69] “Directory | Coest.” [Online]. Available: http://coest.cstcis.cti.depaul.edu/index.php/tracelab/component-directory. [Accessed: 12-Feb-2013].
[70] D. D. Nardo, N. Alshahwan, L. C. Briand, and Y. Labiche, “Coverage-Based Test Case Prioritisation: An Industrial Case Study,” in *ICST*, 2013, pp. 302–311.
[71] *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE, 2013.
[72] M. Hays, J. H. Hayes, A. Stromberg, and A. Bathke, “Traceability Challenge 2013: Statistical Analysis for Traceability Experiments Software Verification and Validation Research Laboratory (SVVRL) of the University of Kentucky,” in *Traceability in Emerging Forms of Software Engineering (TEFSE), 2013 International Workshop on*, pp. 90–94.
[73] “TraceLab.” [Online]. Available: http://coest.org/index.php/about-coest8/current-projects/tracelab. [Accessed: 23-Jan-2012].
[74] R. R. Wilcox, *Applying contemporary statistical techniques*. Amsterdam ; Boston: Academic Press, 2003.
[75] R. Baeza-Yates and B. Ribeiro-Neto, *Modern information retrieval*. New York; Harlow, England: ACM Press ; Addison-Wesley, c1999., 1999.
[76] G. Canfora, F. Mercaldo, C. A. Visaggio, M. D’Angelo, A. Furno, and C. Manganelli, “A Case Study of Automating User Experience-Oriented Performance Testing on Smartphones,” in *ICST*, 2013, pp. 66–69.
[77] L. Deng, J. Offutt, and N. Li, “Empirical Evaluation of the Statement Deletion Mutation Operator,” in *ICST*, 2013, pp. 84–93.
[78] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb, “Symbolic Path-Oriented Test Data Generation for Floating-Point Programs,” in *ICST*, 2013, pp. 1–10.
[79] M. Pezzè, K. Rubinov, and J. Wuttke, “Generating Effective Integration Test Cases from Unit Ones,” in *ICST*, 2013, pp. 11–20.
[80] P. Abad, N. Aguirre, V. S. Bengolea, D. Ciolek, M. F. Frias, J. P. Galeotti, T. Maibaum, M. M. Moscato, N. Rosner, and I. Vissani, “Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving,” in *ICST*, 2013, pp. 21–30.
[81] G. M. Kapfhammer, P. McMinn, and C. J. Wright, “Search-Based Testing of Relational Schema Integrity Constraints Across Multiple Database Management Systems,” in *ICST*, 2013, pp. 31–40.
[82] G. Shu, B. Sun, A. Podgurski, and F. Cao, “MFL: Method-Level Fault Localization with Causal Inference,” in *ICST*, 2013, pp. 124–133.

[83] A. Yeolekar, D. Unadkat, V. Agarwal, S. Kumar, and R. Venkatesh, “Scaling Model Checking for Test Generation Using Dynamic Inference,” in *ICST*, 2013, pp. 184–191.
[84] A. Eriksson, B. Lindström, and J. Offutt, “Transformation Rules for Platform Independent Testing: An Empirical Study,” in *ICST*, 2013, pp. 202–211.
[85] L. Yu, Y. Lei, M. N. Borazjany, R. Kacker, and D. R. Kuhn, “An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation,” in *ICST*, 2013, pp. 242–251.
[86] T. Yu, X. Qu, M. Acharya, and G. Rothermel, “Oracle-based Regression Test Selection,” in *ICST*, 2013, pp. 292–301.
[87] M. J. Arafeen and H. Do, “Test Case Prioritization Using Requirements-Based Clustering,” in *ICST*, 2013, pp. 312–321.

Vita

Mark Allen Hays

Place of Birth:

Lexington, Kentucky

Education:

University of Kentucky

Bachelor of Science in Computer Science, December 2009

Professional Positions Held:

IBM, Software Engineer

Jan 2007 – Present

DePaul University, Graduate Student Researcher

June 2013 – August 2013

Galmont Consulting, Graduate Student Researcher

May 2011 – May 2012

IBM, Technical Editing Intern

May 2006 – Dec 2006

Professional Publications:

M. Hays, J. H. Hayes, A. Bathke, "Validation of Software Testing Experiments: A Meta-Analysis of ICST 2013," to appear in Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on, 1-4 April 2014.

M. Hays, J. H. Hayes, A. Stromberg, A. Bathke, J. Cleland-Huang, and A. Czauderna, "Testing in TraceLab: Empirical Framework and Application," University of Kentucky, Lexington, KY, Rep. 527-13, 2013.

M. Hays, J. H. Hayes, A. Stromberg, and A. Bathke, "Traceability Challenge 2013: Statistical Analysis for Traceability Experiments Software Verification and Validation Research Laboratory (SVVRL) of the University of Kentucky," in *Traceability in Emerging Forms of Software Engineering (TEFSE), 2013 ICSE Workshop on*, pp. 90–94.

M. Hays and J. Hayes, “The Effect of Testability on Fault Proneness: A Case Study of the Apache HTTP Server,” in *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, 2012, pp. 153–158.

M. Hays, M. Billau, and P. Reder. "A method for multiplexing the execution of automated test cases in multiple Web browsers requiring automation of the mouse and keyboard on a single computer (“client”) through a centralized test distribution server (“server”)," http://ip.com/IPCOM/000227933, May 2013.

B. Gibson, M. Hays, D. Hays, "Making Charts accessible to persons with disabilities," http://ip.com/IPCOM/000189284, March 2009.