UNIVERSITY OF CALIFORNIA,
IRVINE

Automatic Natural-Language Fault Diagnoses

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Nicholas DiGiuseppe

Dissertation Committee:
Assistant Professor James A. Jones, Chair
Professor Crista Lopes
Associate Professor Ian Harris

2015

DEDICATION

To my God who is my strength, my support and my friend.
To my adoring wife who supported me and my dreams through long years of school and children and internships with new babies.
To my children, who will likely have no memories of these years, but who bore many difficulties so I could finish school.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank the NSF for their generous support. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0808392.

I also wish to thank my advisor, James Jones for the years of guidance and direction, for helping refine my ideas, and generally for being a great mentor.

I finally wish to thank my lab mates who read and reviewed my papers, who helped my prep for conferences, talks, and a whole host of other things and generally have been great friends these few years.

CURRICULUM VITAE

Nicholas DiGiuseppe

EDUCATION

Doctor of Philosophy in Software Engineering **2015**
University of California, Irvine *Irvine, California*

Masters of Science in Software Engineering **2013**
University of California, Irvine *Irvine, California*

Bachelors of Science in Information and Computer Science **2007**
University of California, Irvine *Irvine, California*

RESEARCH EXPERIENCE

Graduate Research Assistant **2010–2015**
University of California, Irvine *Irvine, California*

Undergraduate Research Assistant **2005–2007**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2010–2011**
University of California, Irvine *Irvine, California*

Master Tutor **2005–2006**
University of California, Irvine *Irvine, California*

REFEREED JOURNAL PUBLICATIONS

Fault Density, Fault Types, and Spectra-based Fault Localization **2014**
Journal of Empirical Software Engineering

SWEET Ontology Coverage for Earth System Sciences **2014**
Journal of Earth Science Informatics

Security Testing of Session Initiation Protocol Implementations **2009**
International Journal of Information Security

Real Genders Choose Fantasy Characters: Class Choice in World of Warcraft **2007**
First Monday

REFEREED CONFERENCE PUBLICATIONS

Automatically Describing Software Faults **Aug 2013**
International Symposium on the Foundations of Software Engineering

Semantic Fault Diagnosis: Automatic NaturalLanguage Fault Descriptions **Nov 2012**
International Symposium on the Foundations of Software Engineering

Concept-Based Failure Clustering **Nov 2012**
International Symposium on the Foundations of Software Engineering

Software Behavior and Failure Clustering: An Empirical Study of Fault Causality **Apr 2012**
International Conference on Software Testing, Verification and Validation

Fault Interaction and its Repercussions **Sep 2011**
International Conference on Software Maintenance

On the Influence of Multiple Faults on Coverage-Based Fault Localization **Jul 2011**
International Symposium on Software Testing and Analysis

INTERSTATE: A Stateful Protocol Fuzzer for SIP **Aug 2017**
Defcon

ABSTRACT OF THE DISSERTATION

Automatic Natural-Language Fault Diagnoses

By

Nicholas DiGiuseppe

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2015

Assistant Professor James A. Jones, Chair

The overall debugging process is a complicated and troublesome task, involving several stages and dimensions of human comprehension. Developers seek understanding of several aspects of faults, such as, *where* the faults are located in the code, *what* sequences of actions invoke faults that cause failures, and *why* the program is failing due to the faults. Despite a large body of research for providing automation for the first two tasks, very little work has been conducted in helping to assist in the last question of "why" — that is, for describing the nature of the fault. I propose an automated approach to *describe* software faults that can indicate the nature of faults and their failures; thus ameliorating comprehension and reducing manual effort. To create this automated approach I propose using a combination of dynamic analysis techniques with information retrieval and text mining to generate natural language clues. In this document I outline the design of my technique along with a research plan that I have used to investigate the effectiveness of using a such an approach. In particular, I detail five evaluations that provide a thorough assessment of the relative merits and shortcomings of the proposed technique.

Chapter 1

Introduction

The software debugging process is composed of multiple steps, each of which is dominated by developer-comprehension tasks (*e.g.,* recreating the failure, interpreting the failing behavior, identifying what would have been the correct behavior, describing the failure for a bug report, narrowing the search for the locations in the program that are faulty, understanding what it is doing wrong, identifying the changes that need to be made to correct the fault, and implementing the fix).

In fact, Gilmore studied 80 developers who performed debugging tasks, and found that, "the success of the experts at debugging is not attributed to better debugging skills, but to better comprehension," [30]. In addition, von Mayrhauser and Vans performed observational field studies of professional maintenance programmers, and found that program understanding was of central importance during debugging, and that misunderstandings by the programmers caused protracted debugging sessions [78]. More recently, Bettenburg *et al.* found that well-described bug reports help comprehending the problem better, consequently increasing the likelihood of the bug getting fixed [9]. In essence, these studies suggest that before faults can be fixed, they first must be understood.

A well-explored area of research to assist in these debugging comprehension tasks is that of *fault localization*, which assists in the task of finding *locations* for the faults. Some notable and popular fault-localization techniques that have been proposed by researchers include Weiser's work on automated program slicing for debugging [80], Agrawal *et al.* 's work on dynamic slicing for debugging [3], and Zeller *et al.* 's work on localizing cause-effect chains [85]. More recently, a significant field of research has emerged to develop and study a new family of fault-localization techniques based on correlation of execution events with program behavior (commonly referred to as "spectra-based," "spectrum-based," "coverage-based," "correlation-based," or "statistical" fault-localization techniques), (*e.g.,* [7, 41, 46, 47, 84]). Further, researchers have extended such techniques to provide more context to these locations in the code, thus providing sequences of locations that may describe a fault (*e.g.,* [12, 36, 38, 54]).

Although each of these techniques assist comprehension to support debugging, they all share a common feature: they address the "where" question, *i.e.,* "*where* is the fault located?" They provide structural information to the developer, but still require the developer to inspect these areas to then understand the functionality and domain of the faults. None of these techniques inherently assists with the other comprehension tasks required of the developer, such as "why is the program failing?" or "what is the nature of the fault?" or even "why these locations?" And, although techniques that provide common execution patterns (*e.g.,* [38]) can assist comprehension, all such techniques present their results with program locations and structure rather than a semantic description. Thus the main shortcoming of existing techniques is that they do not directly attempt to alleviate the cognitive burden of developers during debugging.

To address such descriptive-understanding questions, to provide some automation to *describing* the faults, and directly reduce the cognitive load of developers during debugging I propose the following hypothesis:

Automatically generated natural-language descriptions of a fault (*i.e.,* Semantic Fault Diagnoses) will assist developer comprehension during the debugging process.

To create these descriptions, I propose a technique that mines the source code of the program and performs analyses that identify the topics describing the nature of the faults. Such techniques, which I am naming "semantic fault-diagnosis techniques" (or *SFD techniques*), provide automatic natural-language cues to the developer about the domain topics that are likely descriptive of the faults and their failures.

The technique works by first mining the source code for words that were embedded in the code in the form of identifiers (*e.g.,* variables, methods, classes) and developer comments, while in parallel identifying a likelihood of faultiness for every location in the source code, and then finally analyzing the extracted words to identify those words that occur frequently throughout the most-likely faulty locations but least frequently in the least-likely faulty locations. As such, the technique is likely to extract descriptive domain-relevant words such as "tire," "brake," "ABS," and "skid" and unlikely to extract non-descriptive programming-language words such as "if," "for," and "exception."

I foresee use and benefit of such SFD techniques at multiple phases during software maintenance and debugging. Some example semantic fault-diagnosis use-cases include: immediately after test failure when attempting to understand why the program failed or as part of an automated test-suite run (for example, as a overnight build-and-test cycle). The diagnoses are designed to provide an easy and automatic way of summarizing the nature of the faults that were encountered. As another use-case scenario, a developer could use a SFD tool in conjunction with more traditional fault-localization tools in order to more quickly comprehend the debugging task.

To provide an exploration of the potential impact brought by such a technique, I view *com-*

prehension of faults as the central characteristic to be assessed. While a more thorough treatment of my hypotheses are presented in Chapter 3, I briefly present the five evaluation types here. I propose the following five evaluations: (1) a quantitative evaluation leveraging existing fault descriptions, (2) a qualitative evaluation via assessments from real developers, (3) a comparative study between SFD and existing locational based techniques, (4) an evaluation to determine the correlation between each step in the SFD process and the overall results, and (5) a user study to evaluate any practical impact of use during debugging.

Evaluation 1 is designed to quantitatively compare the natural-language descriptions automatically generated by SFD against natural-language descriptions manually generated by humans attempting to describe the fault. Evaluation 1 explores such questions as, "does SFD produce descriptions *similar* to those written by real developers?"

Evaluation 2 is designed to qualitatively assess the quality of the descriptions automatically generated by SFD. Evaluation 2 explores such questions as, "does SFD produce *quality* descriptions that accurately describe the bug?"

Evaluation 3 is designed to assess the relative difference between using automatically generated SFD and state-of-the-art locational based techniques to understand faults. Evaluation 3 explores such questions as, "does SFD enable more complete, faster, or more helpful information compared to existing automated debugging tools?"

Evaluation 4 is designed to determine the correlation of each component within the SFD workflow with the final result. Evaluation 4 explores such questions as, "what is the impact of each component of the SFD workflow upon the final result?"

Finally, evaluation 5 is designed to assess the impact of using SFD in practice. Evaluation 5 explores such questions as, "does SFD actually improve developers performance while debugging, and if so, how does it help?"

In this paper, I propose my insight and the motivation behind this proposal along with envisioned usage scenarios and a motivating example (Chapter 2), next is an explanation of the hypotheses addressed by this research plan (Chapter 3), then is my methodology to SFD's configuration and implementation options (Chapter 4), next is the potential benefits and limitations of the SFD framework (Chpater 5, then my evaluation strategy is presented at a high level for all hypotheses (Chapter 6), this is followed by the work I have completed to date broken down by major hypothesis (Chapters 7, 8, 9, 10, and 11), next I present the related work to my technique (Chapter 13), and finally I present the conclusions and contributions of this research agenda (Chapter 14).

Chapter 2

Motivation

SFD is motivated in part by the practice of real developers; when a fault is encountered, a textual description is written to communicate the problem clearly to other developers (*i.e.,* a bug report). As such, the goal of SFD is to leverage the ample context that is already encoded by developers in the source code to create a similarly valuable textual description.

2.1 Natural-Language Source-Code Mining

Many computer science and software engineering researchers have identified the wealth of semantically rich information that is encoded in the source code: in the logic and data structures, in the identifier names, and in developer comments. An early example of the power of the source code to influence programmer comprehension was Dijkstra's letter "Go To Considered Harmful," in which he describes the difficulties for programmers attempting to understand and maintain unstructured code [24]. Shortly thereafter, Dahl, Dijkstra, and Hoare describe early efforts to make source code more comprehensible through structured programming [14]. In that early work, Dijkstra describes the benefits of abstracting func-

tionality through proper naming of the abstractions. Shneiderman reports on his research and empirical studies that demonstrated that "commenting, mnemonic variable names and modular program design had a significant impact" on program comprehension [72]. Letovsky presents observational studies of programmers that show that their subjects held the cognitive rules such as "variables and data structures are named for the data they hold" and "routines are named for the functions they perform" [45]. Rajlich and Wilde recognize the rich semantics that are encoded in identifiers and describe the role of concepts in program comprehension, and describe how developers utilize source code and their understanding of concepts to perform software maintenance tasks [66]. They describe developers' attempts to find features based on source code identifiers (*e.g.,* "when searching for the location of cut-and-paste, the programmer may want to search for identifiers 'cut', 'cutPaste', etc.").

2.1.1 Source Code Encoding of Both Developer Intention and the Potentially Fallible Reality

Specifically for the debugging task, developers must understand not only what the program *is* actually doing, but also what it *should be* doing. Biggerstaff *et al.* examined this difference between the implementation reality and human intention [10]. They explain that the source code provides not only a definition of the true, current structure and semantics of the program (*i.e.,* what the software *is*), but also includes semantic intent (*i.e.,* what the software *should be*). The source code comments often express the intent of the developers. In addition, variable and method identifiers can also provide expressive intent.

Biggerstaff *et al.* assert that, "both forms of the information must be present for a human to manipulate programs. . . in any but the most trivial way." They are not alone in this assertion. Etzkorn *et al.* find that, "from the computer code *what* task is being done can be determined, but it is only from the comments *why* that task is being done can be understood," and, "it is

more important to understand the comments than to understand the computer code," when modifying a system [27]. More simply, if a developer wants to modify code in any nontrivial way, they require executable code and comments to gain a sufficient understanding of what the code is doing, and what it should do. Vinz *et al.* [77] and Marcus *et al.* [53] agree that without both types of information to form a proper concept of the system, developers are handicapped when attempting to modify it. And while it is well known that comments can drift in their correspondence to the source code, in many cases they provide valuable insight as to the original developer intention.

2.2 Example Debugging Scenario

To illustrate the potential benefits of semantic fault diagnosis, I present a scenario based on a real program along with actual results from our SFD prototype, MANTIS. The example program is an implementation of the TETRIS game. In this game, players guide falling shapes to form complete rows. Once a row has been completed — that is, there are no empty gaps in the row — the program should remove that completed row.

Consider a developer who arrives at work after a nightly build and automated testing. She finds the test suite produced more than 50% test failures, and as such, the test-case names are not be helpful in assessing the nature of the fault. Indeed, the test case names and descriptions are likely not precise enough to describe the fault.

In addition to the typical text-based report of test-case results and failures, the developer is presented with the SFD results shown in Table 2.1. This table displays a list of the top terms that were identified as relevant to failure-execution-correlated code. Note that this table presents terms that are automatically extracted from the program utilizing a semantic fault-diagnosis technique (N.B., these are actual results of our prototype, MANTIS).

Table 2.1: Results of a semantic fault diagnosis from variables, method names, and developer comments.

xPos++	figure.moveLeft()
KeyEvent.VK_LEFT	e.getKeyCode()
moveLeft()	canMoveTo()
"left"	"Moves the figure one step to the left"
left	"Handles a keyboard event"

With such results, she can quickly gain a basic understanding of the program features that are implicated in the faults causing failures. Such a basic understanding gives her an early indication of (1) *why* the program is failing, (2) the nature of the faults, (3) the importance of such failures, and (4) who may be appropriate to debug such failures, in order to help guide future investigations and code examinations. For the given example diagnosis in Table 2.1, this developer likely infers that the fault involves functionality intended to "handle a keyboard event" and "moves the figure. . . left."

While our example developer does not yet know the exact fault location or improper logic, she has an intuition of the nature of the problem, prior to inspecting other, more technical and detailed evidence such as stack traces, execution-coverage reports, and testing logs. And while the later stages of debugging, such as fault localization, might still be necessary, such an automatic, early diagnosis of this fault — when the left key is pressed, the program behaves incorrectly — may inform those tasks.

As a demonstration that current techniques are insufficient for this fault-comprehension task, consider the same scenario except the developer only has the test results, the fault localization results, and the feature extraction results.

Consider first Figure 2.1, which displays the actual JUnit results from this particular test suite. Examining the test-case names suggests that the only the basic game over feature seems to be the working, as a test involving *all* other keys or features fail. These results are clearly not precise enough to pinpoint a particular fault.

Figure 2.1: Actual JUnit results from the Tetris game with a single fault.

Table 2.2: Top-12 results from fault localization.

Source Code	**Suspiciousness**
if(e.getKeyCode() == KeyEcent.VK_P)	100
if(figure == null || moveLock || thread.isPaused())	100
return board != null	100
if(isAttached())	100
xPos = 0;	100
yPos = 0;	100
newX = board.getBoardWidth() / 2;	100
for(i = 0; i < shapeX.length; i++) {	100
this.board = board;	100
if (!canMoveTo(newX, newY, orientation)) {	100
board.update();	100
switch(e.getKeyCode())	100

Next, examine Table 2.2, which displays the actual fault localization results for this test suite. These lines are in disparate locations in the code and only seem to point to a general problem in the board, and potentially the movement of figures. A developer would have to spend considerable time trying to understand the context of each line (i.e., context switching) and attempting to ascertain the nature of the problem.

Lastly, consider feature extraction. Unfortunately, this technique can only provide a general topic modeling of the system unless the developer inserts a specific query. Indeed, it either provides high-level, general information, or requires the user to have an existing understanding of the bug before it can "extract" any features.

While simple, this example clearly demonstrates the potential held within a technique that can provide natural language clues to developers. I thus propose *Semantic Fault Diagnosis* (SFD) to (1) address existing limitations in current program-comprehension and fault-localization techniques (inability to describe a fault and descriptions are entirely locationally or structurally based respectively), and (2) fill the vacuum regarding to fault-comprehension techniques. I define SFD as a framework or a family of techniques that extracts strings

and terms from source code in order to generate descriptions of test-case failures. SFD only requires three inputs: pass/fail data for a test suite, instrumentation data for said test suite, and the source code. As such, SFD can be leveraged immediately after a software failure is encountered, meaning it does not require a bug report and the developer requires no preexisting knowledge of the bug or the failure. A more complete treatment of what SFD is given in Chapter 4, and will be considered after the hypotheses.

Chapter 3

Hypotheses

The research agenda that I implemented is based upon and extends the state of the art as presented in Chapter 13. More concretely, my research agenda explores the potential value in leveraging an automatic technique to generate natural language fault descriptions and their ability to aid developer comprehension during debugging.

My research agenda investigates the following hypothesis:

Hyp: Automatically generated natural-language descriptions of a fault (*i.e.*, Semantic Fault Diagnoses) describe faults with similar key words as manually-written-developer descriptions, produces fault specific and helpful descriptions, reduces the cognitive burden more than existing techniques, contains an optimizable workflow, and can improve a developer's chances to find and fix a fault.

In order to investigate *Hyp*, I propose investigating the following sub-hypotheses, which when considered together enable sufficient understanding and context to answer *Hyp*.

- **H1**: Semantic Fault Diagnoses describe faults with similar key words to those written manually by real developers.

  H1 is related to the precision of word choice that a fault description must have to be semantically valuable. The goal of SFD is not to replicate a bug report or even provide grammatically correct sentences, indeed, doing so is beyond the scope of this proposal. However, one way to assess whether SFD results assist developer comprehension is to identify whether they use the same terms/key-words that developers use when they manually describe the fault. This comparison is meaningful because existing bug reports for closed bugs (faults that have been fixed) are likely valuable in their descriptive power to elucidate understanding regarding the fault they describe and if SFD uses similar terms/key-words, then there is some evidence that the SFD result would be comparably valuable in its descriptive power.

- **H2**: Automatically generated natural-language fault descriptions produce quality descriptions that accurately describe the fault.

  - $H2_1$: *Does Semantic Fault Diagnoses contain words that are descriptive of the fault?*
  - $H2_2$: *Is Semantic Fault Diagnoses helpful in understanding the fault?*

  H2 recognizes that a description needs to be more than describe the domain of the software in question. Consider a compiler bug report, which likely contains terms like: "compile", "binary", or "code". However, a fault description relaying only these types of words, which describe the program domain but are not exclusive to the fault, fail to have the semantic power to assist developers. As such, there are two research questions that enumerate the space of potential descriptive words to specifically address the semantic "quality" of a description.

- **H3**: Semantic Fault Diagnoses reduce the cognitive burden when attempting to understand a fault more than existing location-based debugging techniques.
  - $H3_1$: *Is it less time consuming to interpret a Semantic Fault Diagnosis than a location based fault description?*
  - $H3_2$: *Do developers gain a clearer understanding of a fault from a Semantic Fault Diagnosis than from a location based fault description?*
  - $H3_3$: *Do developers find Semantic Fault Diagnoses more "helpful" to understand faults than a location based fault description?*

H3 is a comparative looking hypothesis that explores whether this new technique gives added benefit when compared to existing locational based techniques. While existing research has clearly indicated that there is added value in mining the source code for semantic information (see Section 13), this hypothesis explores specifically whether this is true for the debugging problem space. Unfortunately, "cognitive burden" is not a trivial aspect to measure, and as such for the purposes of this hypothesis, it is assessed leveraging three distinct measures which each seek to approximate a different aspect of "cognitive burden". As such, *H3* contains three research questions, one for each of these measures. The first is strictly quantitative, how fast developers are able to interpret each fault description; in essence, if developers are able to consistently interpret one type of result more quickly, I speculate that it is because it has less cognitive overhead. The second is qualitative, and assesses the resulting understanding a developer gained after leveraging either SFD or locational based techniques; if developers are able to consistently form a more correct understanding with one type of result, I speculate that it is because it improves their cognitive understanding. The third is also qualitative and leverages feedback directly from the developers as to which technique they found more helpful; if developers consistently find one result type more helpful I speculate it is more beneficial at reducing cognitive burden.

- **H4**: Optimizing any step of the natural-language pipeline or fault localization within Semantic Fault Diagnosis similarly improves the results.
  - $H4_1$: *Does optimizing text parsing provide higher quality Semantic Fault Diagnoses?*
  - $H4_2$: *Does optimizing text normalization provide higher quality Semantic Fault Diagnoses?*
  - $H4_3$: *Does optimizing dimensional-reduction techniques provide higher quality Semantic Fault Diagnoses?*
  - $H4_4$: *Does optimizing the fault localization produce higher quality Semantic Fault Diagnoses?*

  H4 is related to understanding the correlation between components of SFD's workflow and the final result. This hypothesis and its corresponding four research questions explore how the use of various natural language techniques in SFD impacts the data as it moves through the pipeline, and how its dependence upon fault localization impacts its results. In essence, by this stage there will be existing evidence to demonstrate that SFD works and this hypothesis seeks to better understand the *why* and *how* of its success.

- **H5**: Semantic Fault Diagnoses improve developer performance while debugging.
  - $H5_1$: *Do developers leveraging Semantic Fault Diagnoses will find the fault more often?*
  - $H5_2$: *Do developers leveraging Semantic Fault Diagnoses fix the fault faster or more often?*
  - $H5_3$: *Are there differences between debugging complex and simple programs in terms of improvements imparted by Semantic Fault Diagnoses?*

H5 explores the practical value of using SFD during the debugging process. While previous hypotheses approximated value and explored important facets of how/why SFD may reduce cognitive burden, an evaluation of a debugging technique would be insufficient without investigating whether said technique actually makes a difference in practice. As such, *H5* and its corresponding research questions consider two different aspects of the debugging process, finding the fault and fixing the fault. Along with these two questions, $H5_3$ specifically considers the impact of SFD on programs of differing complexity.

When considered as a whole, these hypotheses build upon one another with increasing evidence that culminates in ascertaining whether SFD reduces the cognitive load placed upon developers during the actual debugging process. Further, each hypothesis explores a different facet of the fault descriptions provided by SFD to enable a more complete understanding of both its value, and impact.

Chapter 4

Semantic Fault Diagnosis Methodology

My approach involves seven steps (as shown in Figure 4.1) that can all be completely automated. I describe this approach step by step in the following subsections (as previously described in [22]).

Step 1. Instrument Code: In the first step, the program's source code is instrumented so that execution information can be captured and recorded. The instrumentation may gather any type of coverage or profiling information — so long as it can be used for fault localization — including commonplace and lightweight statement instrumentation built into common software-testing tools. I chose to instrument the source code using lightweight and commonplace statement instrumentation, such as those used to measure testing adequacy. My implementation of SFD uses Cobertura for Java based programs and Gcov for C programs.

Step 2. Run Test Suite: In the second step, the test suite (or any valid subset containing

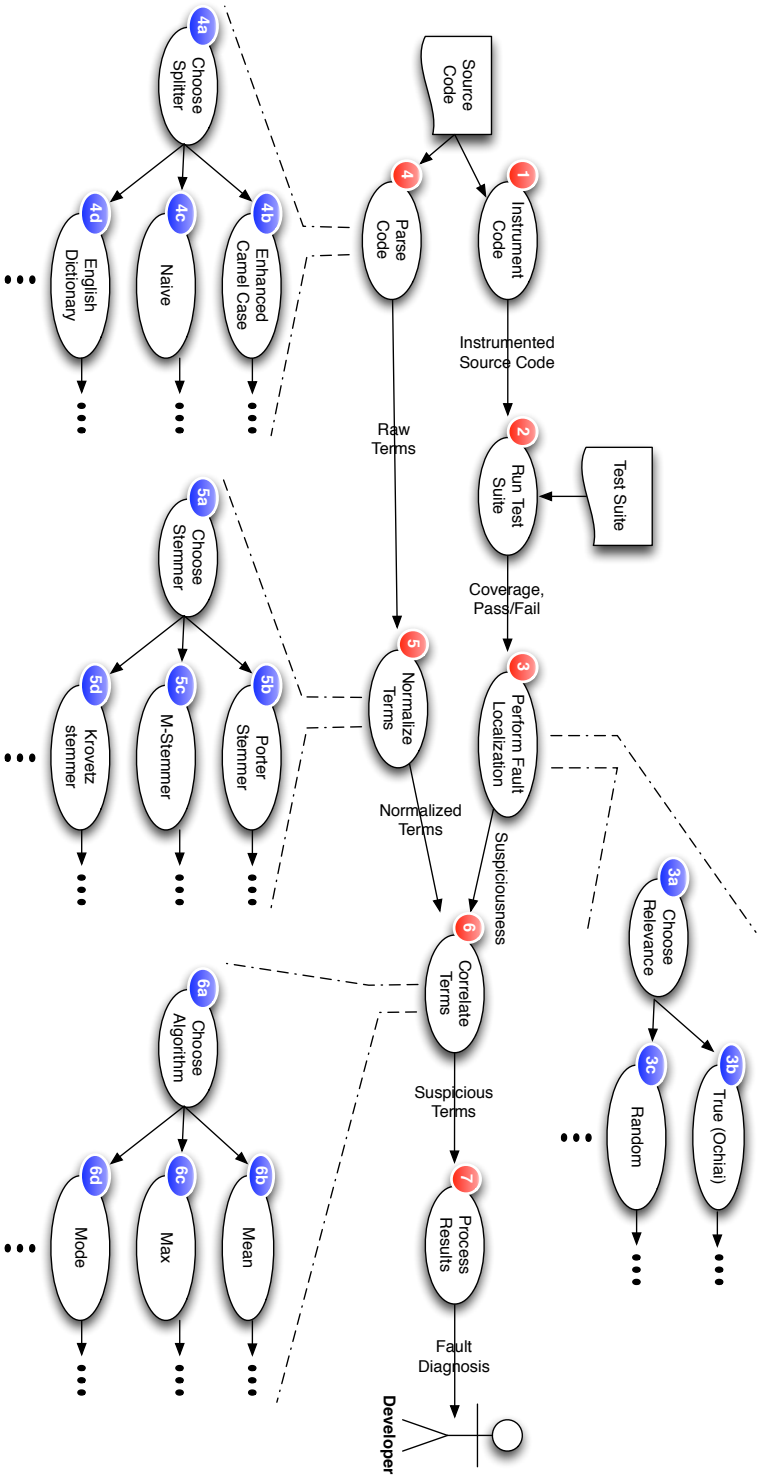

Figure 4.1: A process diagram depicting the seven steps for performing Semantic Fault Diagnosis with various parameterization options.

at least one passing and failing test case) is run against the instrumented version of the source code. If any test cases fail, the pass/fail status and the coverage information for each test case are saved.

Step 3. Perform Fault Localization: In the third step, the information gathered from the testing is used to perform fault localization. Many different approaches for fault localization may be used. However, approaches that provide continuous, variable measures of the failure-correlation of code can be particularly useful. I performed fault localization with the Tarantula [39] tool using the Ochiai [1] suspiciousness metric.

Step 4. Parse Code: In the fourth step, the source code is parsed to extract strings, terms, words, and expressions. The parsing can be performed in a number of ways and include a variety of elements, however, some of the richest and most meaningful strings include the developer comments in the source code. Identifiers such as variable, method, class, and file names can also contain meaningful information about the purpose of the code in which it is involved.

Identifiers may also be further parsed to extract meaningful substrings. For example, consider the method name "`getReducedCost()`" which can be further parsed to its individual words, "get," "Reduced," and "Cost."

The parsing of the source code is performed simply by scanning the source code files and separating strings by the following delimiters: {, }, space, tab, `,`, `?`, `!`, `;`, `:`, `@`, `'`, `*`, `(`, `)`, `/`, and \. Strings are partitioned by spaces, tabs, and punctuation. Additionally, SFD splits identifiers by recognizing *camel case*: the use of capitalization to create compound word identifiers.

Step 5. Normalize Terms: In the fifth step, the strings extracted in the fourth step are normalized to account for variations in the word use. Variable tense, capitalization,

suffixes, and synonyms are among the characteristics of the strings that can be normalized. For example, "Reduced" can be normalized to "reduce" so that it can be recognized as the same meaning as other words such as "reducing." This process is commonly referred to as as *stemming.* In addition, common natural-language words, such as "the" and "of" can be removed. These words are often referred to as *stop words.*

To normalize the words that are identified, SFD first filters the list by removing stop words and then perform stemming and set all capitalization to lower-case. SFD, leverages the Porter stemmer [61]. I preserve expressions such as the dereferencing of a type, as well as its parsed form. For example, the expression "`x = myResult.getInfo()`" would be parsed into the following strings: `x`, `my`, `result`, `myResult`, `get`, `info`, `getInfo`, and `myResults.getInfo`. Note that I do not normalize the un-split versions of the words (*i.e.,* I do not normalize the term `getInfo`). Thus, each expression yields both its normalized terms and the non-normalized originals. Developer comments in the source code are both preserved and parsed as well to allow for phrases or sentences that can be more expressive than the individual terms.

Step 6. Correlate Terms: Once the terms have been normalized and the fault-localization has been performed, the normalized terms are correlated with the localization results. Terms that frequently occur in code that is strongly correlated with failure are identified as "suspicious" terms. In contrast, terms that occur throughout the codebase (*i.e.,* both correlated and uncorrelated with failure) are considered as less suspicious than the terms that only occurred in the strongly correlated code.

Mantis assigns "suspiciousness" values to the extracted terms by leveraging the suspiciousness results from the fault-localization technique. The Tarantula implementation that I use assigns a suspiciousness value to each statement in the program according to its execution's correlation with test case failure. Statements whose execution strongly correlate

with failure are assigned high suspiciousness values, and statements whose execution do not correlate with failure are assigned lower suspiciousness values. In other words, if test cases regularly fail after a particular statement is executed, and test cases rarely pass after executing that same statement, the statement is considered highly suspicious of contributing to the *cause* or *fault* of those failures.

MANTIS assigns suspiciousness values to the extracted terms by taking the arithmetic mean of the statement suspiciousness values from the statements where those terms occurred. For example, if the term "refresh" was extracted from parsing statements whose suspiciousness values were 0.9, 0.8, and 0.2, the suspiciousness value for the term "refresh" would be $(0.9 + 0.8 + 0.2)/3 = 0.63$.

However, when considering applying such a fault localization technique to the words within the statements, such approach would only work for terms that are found in executable statements, because those are the only lines that are assigned suspiciousness values by fault localization techniques. Because developer comments in the source code are a particularly strong indication of developer intentionality [10] (as previously discussed in Section 13), MANTIS also assigns suspiciousness values to those lines (and their corresponding terms). MANTIS's approach for assigning suspiciousness values to the developer comments is to estimate the code that is being documented by the comment and then assign the arithmetic mean of the suspiciousness values of those lines. The MANTIS tool, estimates the relevant code for a comment as the $2n$ lines that follow an n-line comment (my motivation for this is discussed later). For example, if a comment were contained to a single line, I take the mean of the suspiciousness values for the next two lines. If a comment were three-lines long, I would take the mean of the suspiciousness values for the next six lines. I assign this averaged suspiciousness value to the entire comment, which can then be used to inform the suspiciousness values for each of its composite terms. I do not perform this step for comments that occur at the end of a line (*e.g.,* "`i++ // increment`") — for these, the suspiciousness

of the comment is the same as the suspiciousness of its statement.

Using the following $2n$ lines was selected because an anecdotal investigation revealed that this $2n$ method seemed reasonable, and formal explorations of more complex comment association heuristics are outside the scope of this research agenda. I chose this approach for estimating relevant lines based upon anecdotal evidence: longer comments often described more steps in the computation, which is approximated by more lines of code. This estimation is made to enable research in this new field as answering this question regarding comments and code is outside the scope of this agenda. Indeed, an investigation ascertaining a general heuristic that describes the correlation between developer comments and the code they specifically describe is a non-trivial task, and while early research has begun to explore this space (such as Vinz. [77]) there is insufficient evidence to recommend a single "best approach" or technique. As this space is explored in the future, improvements to MANTIS can be implemented to better relate comments to code.

Step 7. Process Results: The final step involves processing the suspicious terms and presenting them to the developer. For example, the results can be categorized by the type of their origin (*e.g.,* variable name, comment). The suspicious terms can be sorted and the top terms presented to the developer (*e.g.,* Table 4.1), or the results may be visualized and correlated with the locations in the program.

Table 4.1: Top-3 results from each category of a semantic fault diagnosis.

Words	**Category**
yPos– – figure.moveAllWayDown() keyEvent.VK_DOWN	expressions
moveAllWayDown handleButtonPressed isControlDown	method names
Moves the figure all the way down limit down	comments

As a component of MANTIS, (and after feedback from pilot studies) I implemented a tag cloud to automatically highlight the most failure-relevant terms that compose the fault diagnosis. I use a tag-cloud implementation of the most failure-correlated terms that compose the fault diagnosis. I call the visualization of these fault diagnosing terms, a *Buggle*. An example Buggle is shown in Figure 4.2. The size of a term is based on its frequency within the results and its overall score. For the layout algorithm, I use the open source plugin JQCLOUD,[1] which places the most important word in the center and the remainder in an elliptical shape around the center.

The correlations technique, is based upon the open source tool `Concordle`.[2] Figure 4.3 demonstrates the correlation features of Buggle that occurs when the word "down" is clicked upon in Figure 4.2. Down, along with any word which correlates to it, remain solid while all other words become mostly transparent. Also notice that at the bottom of Figure 4.3 there is a concordance. This section only becomes active when a word is clicked upon from the cloud, and displays all words which correlate to the clicked upon word (just like the could). However, the concordance additionally displays all words which are correlated but not suspicious enough to be present in the Buggle's visualization.

Correlation is currently defined as words which are subsets of each other, method names wherein the method body contains the word, all the words inside the method body (if a method name is selected), or developer comment words which are close in proximity to the word (where close is determined by a developer set parameter). While many potential options for determining *correlation* exist, these few are picked for simplicity and responses from users during a pilot study.

In the end, the results are intended to give developer-written, natural-language indications of the features or logic involved in test-case failure, and thus serve as a description of the

[1] https://github.com/DukeLeNoir/jQCloud
[2] http://folk.uib.no/nfylk/concordle_dev/

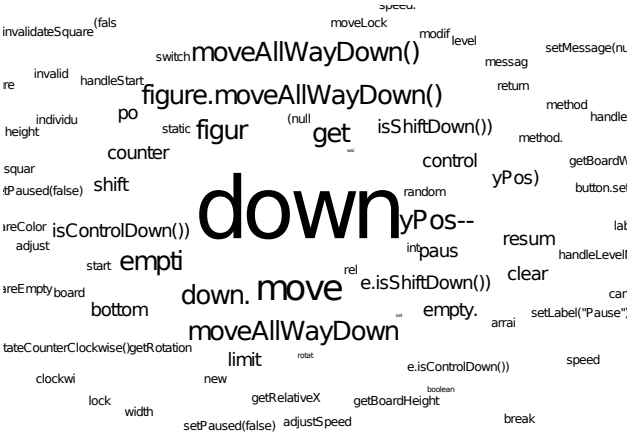

Figure 4.2: Example Buggle for the game Tetris. The fault that this fault-diagnosing Buggle describes occurs when a user pushes the "down" button to move the figure down.

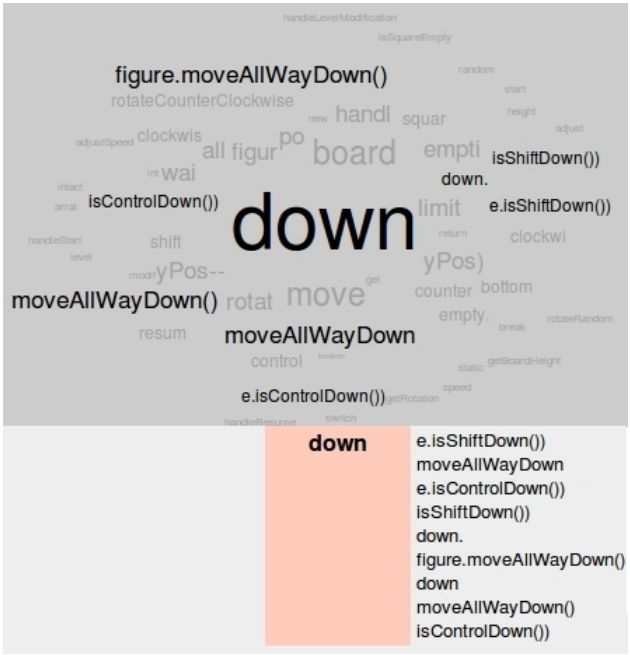

Figure 4.3: An example of the SFD results for the game Tetris showing how the word "down" correlates with the rest of the rest of the code base.

cause of the errors.

Chapter 5

Potential Benefits and Limitations of SFD Methodology

The previously described framework to automatically generate a fault diagnosis contains multiple inherent assumptions concerning its use and implementation. To elucidate understanding regarding SFD, I explicitly discuss these potential benefits and limitations that correspond to the framework as it is described.

One inherent assumption regarding SFD is that the seven step process is used to describe *a single* fault. Unfortunately, it is possible and indeed likely that software will contains multiple faults. A concern then is that SFD will not describe any fault, but a conglomeration of multiple faults that in essence describes no fault but merely the noise between faults. This same concern has been identified with fault localization techniques (e.g., [16, 19, 39, 40, 41, 75, 87]) wherein studies have suggested that the effectiveness declines on all faults as the number of faults increases. Because SFD utilizes fault localization to infer those lines that most uniquely represent the fault, if a fault cannot be localized, the description will likely be poor (and conversely, if the fault can be localized, the description is likely of

quality). Notwithstanding these concerns regarding multiple fault programs, recent studies have explicitly analyzed whether multiple faults reduce fault localization effectiveness and found that surprisingly, they do not [21]. In previous work the author identified that when multiple faults exist in a program, one fault tends to *obfuscate* or overshadow the others [20, 23] such that fault localization techniques successfully identifies a single fault, the *obfuscating* fault [21]. However, due to this *obfuscation*, the fault which is localized is unchooseable. In other words, while fault localization remains effective in the face of multiple faults, it is unchooseable which fault is localized. Further, an anecdotal investigation by the author found this to be the case, *i.e.,* when multiple faults are in a program a single fault is described. Thus for SFD, in the presence of multiple faults, a description of similar quality to the single fault version should be generated regardless of fault quantity, but the fault being described will be unchooseable. It is of note however, that in circumstances where developers are only interested in describing a specifically chosen, a priori fault, the test suite can exclude all failing test cases but the *one* that displays the behavior of interest effectively eliminating this concern.

Another concern might be that due to SFD's reliance upon dynamic information, a large number of test cases will be required, which may or may not exist. This concern is somewhat limited as SFD *only requires* one failing and one passing test case. Indeed, many studies utilizing fault localization leverage a single failure and multiple passing test cases and have demonstrated quality results (e.g., [60, 70]). Thus, in the worst case, where test cases are sparse, the primary concern is generating a passing result. Without a failure there is no need for this technique, meaning there will be a failing test case (even if said test case is informal). However, in programs where the fault causes *all* functionality to fail (e.g., a problem in the main), SFD cannot be utilized. Thus, one limitation is that if the program cannot be run successfully, SFD cannot be leveraged.

The two primary concerns with the natural language components of SFD are its reliance

upon comments and its assumptions of meaningful naming conventions. As discussed in Section 13 developer comments are leveraged to infer developer intent for the description. However, there is no guarantee that the developers use comments, that existing comments are up to date, or that comments are meaningful. While the previously cited studies demonstrate the value of comments, it may be for a particular project, comments are not used. In this circumstance, SFD will at best describe what the code is doing incorrectly, but be unable to describe what the code should have done. Due to the common practice of using comments is many circumstances (even if not in all circumstances) it is likely that some intent can be inferred from the code, however, a formal study of this topic (comment's value degradation over time) is outside the scope of this research agenda.

Regarding meaningful naming conventions, SFD does not assume that each variable and method name is descriptive, but instead makes two assumptions: one, that the names are somewhat descriptive of the domain, and two, that variable names are consistently repeated across methods. While the first assumption seems reasonable, if most names in a given program are completely undescriptive (*e.g.,* `i, aaa,` or `myVar`), SFD will be unable to provide a fault description, as its results will be reduced to a listing of non-descriptive words from the source. The second assumption is related to the way that terms are correlated within SFD. Imagine that in the majority of methods for a piece of code, the variable `dataArray` is used, and in one of those methods, it is involved with the fault. This type of situation would make it difficult for SFD to determine the relevance of `dataArray` as it appears so frequently in other, non-suspicious areas of the code, and even if it were presented to developers, it would be unclear in which content the suspicious use came from. However, based upon anecdotal experience, this poor naming convention is not practiced by developers, making this assumption also reasonable.

A final potential problem with the natural language aspect of SFD is the potential for information loss. Because SFD only presents a summary of information from the source,

it primarily attempts to provide the *keywords*. However, it is possible that a fault is more complex than can be described using keywords, or that using keywords only captures one facet of a fault. In these cases, the summary will result in information loss, such that the summary may confuse or mislead the developer. The likelihood of this event, and the impact thereof is partially investigated in Experiment five.

Chapter 6

Evaluation Strategy

To evaluate these fault descriptions and answer the aforementioned hypotheses, I propose five experiments (one for each major hypothesis): (1) a quantitative evaluation leveraging existing fault descriptions, (2) a qualitative evaluation via assessments from real developers, (3) a comparative assessment between SFD and locational based fault descriptions (4) an evaluation to determine the correlation between each step in the SFD process and the overall results, and (5) a user study to evaluate the practical benefit of SFD during debugging.

Experiment 1 leverages large-scale subjects with years of development and existing quality fault descriptions. To perform experiment 1, I propose comparing the bug reports from closed and fixed faults to the descriptions generated by SFD. In this comparison, I will evaluate the *precision* of SFD results against the bug report text; I assume the bug report text is an accurate and quality description for the fault (i.e., I use the bug report as an oracle).

While Experiment 1 allows for a generalizable, automated and quantitative evaluation of SFD, it cannot measure the semantic power of SFD results to describe a fault. For example, consider a diagnosis for a failing compiler that contains words like `parse, scan, rules,` or `compile`. This diagnosis would likely have a high precision as those words are typically

used when talking about a compiler even if they are unrelated to the fault.

Experiment 2 leverages real developers to enable the assessment of the semantic power of SFD results to describe a fault and its usefulness in forming a correct understanding. That is, one primary purpose of this experiment is to overcome limitations of experiment 1. To perform experiment 2, I propose generating SFD results for a variety of failures from real programs. Next, allow developers to: examine the code containing the failure, see the failure at run time, modify the code to fix the failure, and generally gain a sufficient understanding of the fault. Then, after the developer has gained a sufficient understanding, provide the developer with the SFD results. The developer can then judge whether the words selected are meaningful (i.e., do they actually describe the fault), whether the words provide a sufficient understanding of the fault, and whether they think the words would be useful to someone actually debugging this fault. In this way, a developer can compare the understanding they gained from actually debugging and fixing the fault against the description from SFD.

This experiment design allows for actual developers to assess the quality and usefulness of SFD results, giving some assurance of their quality. However, because this strategy requires developers to understand each fault prior to measuring the quality of SFD results, it is likely somewhat time consuming and as such, less generalizable.

Experiment 3 allows for a comparison between SFD and existing state-of-the-art automated debugging techniques. I propose a user study where developers attempt to find and describe a bug in source code and answer questions about the relative value of the technique. To help them, developers either get SFD results (and must try to complete the tasks) and then get the locational based results (and attempt to improve their answer) or they start with locational based results and then later gain SFD results. The format of the study is designed to explicitly evaluate the relative benefit of each technique in aiding developers to locate and understand a fault (*i.e.,* enable complete comprehension).

This experiment design allows actual developers to use these results to understand a fault relative to each technique, giving some assurance of which technique better improves comprehension of the fault. However, this experiment does not measure the absolute benefit of each technique, consider that SFD may be better than a locational based technique for understanding a fault, but still not helpful enough to make an impact on actual debugging in practice.

Experiment 4 enables the evaluation of the significance of each step in the SFD process. Indeed, this experiment will attempt to measure the correlation between each step and the end results. I suggest creating multiple SFD results for the same bug using slightly different SFD input parameters (e.g., no normalization, different types of splitting, and different SFL techniques) and evaluating these results just as experiment 2 — this allows for an explicit evaluation of how the quality of the results changed based upon the change in the workflow. This experiment measures the impact and correlation between the various sub-processes in SFD and the end result.

Experiment 5 enables the measurement of the practical benefit of SFD via a user study. For this experiment, I propose an A/B test assessing developers as they debug a program where some developers have access to SFD results and others do not. This type of user study enables the measurement of SFD's impact during the actual debugging process. That is, one primary purpose of this experiment is to overcome limitations of experiment 3.

This experiment design allows for actual developers to use during actual debugging, granting some measure of confidence as to its ability to improve debugging. However, because this strategy requires some set of developers to sit down and debug faults while under observation, it is likely somewhat time consuming and as such, less generalizable.

For simplicity, each hypothesis will first be discussed individually and independently in the following format: their experimental setup, results, analysis, conclusions and the threats to

validity for that study. Following the independent discussion of *H1* through *H5*, a summary analysis of their conclusions when considered as a whole is presented.

Chapter 7

Hypothesis H1

*H*1**:** Semantic Fault Diagnoses describe faults with similar key words to those written manually by real developers.

Because the purpose of semantic fault diagnosis is to assist human understanding, this hypothesis involves real developers through developer-written historical artifacts in bug reports.

7.1 Experimental Design

The summarization of the experimental design, the hypotheses addressed, and the benefits and limitations are given in Table 7.1.

Experiment one attempts to determine how well semantic fault diagnosis techniques perform at extracting and highlighting terms that are relevant to the true diagnoses of faults. To enable such an experiment, I perform the SFD technique on a software system with the following two characteristics: (1) the program contains faults, and (2) the software is accom-

Table 7.1: Summary of Experimental Design.

Hypotheses Addresssed	H1: Semantic Fault Diagnoses describe faults with similar key words to those written manually by real developers	
Object of Analysis	AspectJ (75+KLOCs, 75 bugs)	
Independent Variables	**I1.** Source of code Mining	$i1_a$. Variable Expressions $i1_b$. Method Names $i1_c$. Comments $i1_d$. Combination: $\{i1_a, i1_b, i1_c\}$
	I2. Source of bug report mining	$i2_a$. Initial bug report description post $i2_b$. Final bug report post $i2_c$. Title of bug report post $i2_d$. $\{i2_a, i2_c\}$ $i2_e$. $\{i2_b, i2_c\}$ $i2_f$. All bug report posts $i2_g$. $\{i2_c, i2_f\}$ (*i.e.,* title and all posts)
Dependent Variable	**D1.** Precision	Precision of diagnosis compared with text from fixed historical bug reports.
Benefits	**B1.** Repeatability **B2.** Generalizability **B3.** Automated Evaluation	
Limitations	**L1.** Limited ability to assess comprehensibility of results	

panied by bug reports that were written by actual developers. In this experiment I utilize the popular open source program, AspectJ. Different revisions of the AspectJ source code slightly vary in size, however in all of our used faulty versions, it consists of over 75,000 lines of code, and its test suite contains at least 1,184 test cases. I used the iBugs[1] system to randomly identify 75 real, faulty versions[2], along with their fixes — no mutation or seeding of faults was performed.

Experiment one compares a large quantity of faults' bug reports to SFD results. It provides repeatability (other researchers can perform similar tests with the same software), generalizability (there are a large quantity of real faults with descriptions written by many different

[1]http://www.st.cs.uni-saarland.de/ibugs/

[2]The following versions were used: 28974, 29186, 29959, 30168, 31423, 32463, 33635, 34858, 34925, 36803, 38131, 39436, 39993, 40380, 41123, 41952, 42539, 42993, 43033, 43709, 44272, 46280, 47318, 47754, 47910, 47952, 48080, 49457, 49638, 50200, 51320, 51322, 51929, 52394, 53012, 53981, 53999, 54421, 54625, 55341, 57430, 57436, 59596, 59895, 60015, 61536, 62642, 64069, 64331, 65319, 67774, 68494, 68991, 69459, 70619, 70658, 71723, 72150, 72157, 72528, 72531, 72671, 73859, 73895, 76096, 80060, 80249, 80456, 80920, 81846, 82134, 82752, 84310, 87376, 88652

developers), and an automated evaluation (a simple precision test). With those strengths, also comes a limitation: even for strongly correlated word selection between the developer-written bug reports and the automatically generated fault diagnoses, such correlations are merely proxy measures of actual *meaningfulness* or *comprehensibility* to developers. I address this limitation in other hypotheses.

Independent Variable 1: Source Code mining.

I only evaluate the *top six words* of any diagnosis, meaning that for the diagnosis to be precise, the top words selected by SFD must be relevant to the fault. Although the choice of the quantity of six is somewhat arbitrary, it offers the benefit that is small enough to be quickly interpreted by a developer, and will be motivated below (*e.g.,* three categories of sources). Studies to determine the optimal diagnosis size for human interpretation are outside the scope of this experiment.

Variable Expressions ($i1_a$): Top six words taken from the variable expressions category.

Method names ($i1_b$): Top six words taken from the method names category.

Comments ($i1_c$): Top six words taken from the developer comment category.

Combination ($i1_a$, $i1_b$, $i1_c$): Top two words taken from the: variable expressions, method names, and comments categories (for a total of six words).

Independent Variable 1: Source Code mining.

I mine and parse the related bug reports for each fault from the subject's bug tracking system. The bug-report data includes the title, the initial description of the bug, and each developer comment thereafter in the course of investigating and debugging the fault. For each fault, I use this data from the bug report as an "oracle" (*i.e.,* the *ground truth*) of words that describe the fault and its behavior. Such choice of an experimental oracle is made because the bug reports are historical artifacts that were: (1) written by actual developers

and users who were experienced with the program, and (2) describing the failures and faults. The motivation to include all of the bug report (including the final post) is that for the bug report to be a reasonable oracle, it should describe the fault as completely as possible. For example, the final post often includes an accurate summary of the problem, and while the developers wouldn't have it when fixing the fault in development, MANTIS results are available during development and if the presented words are similar to this final post, then they are presenting the right words to a developer. In short, to help the oracle to be as complete as possible, all aspects of the bug report are included.

Initial bug report description post ($i2_a$): The first post that was submitted as the initial report.

Final bug-report post ($i2_b$): The final or closing post of the report.

Title of bug report ($i2_c$): The title as of the time of the closing of the bug report.

Initial bug report description post and title of bug report($i2_a$, $i2_c$): A combination of first post and title of the report.

Final bug report post and title of bug report ($i2_b$, $i2_c$): A combination of the final or closing post and the title of the report.

All bug report posts ($i2_f$): A combination of all posts.

Title of bug report and all bug report posts ($i2_c$, $i2_F$): A combination of all posts and the title (*i.e.,* the entirety of the bug report).

Dependent Variable 1: Precision.

To evaluate a SFD technique's output against the bug reports, I calculate the precision of the *top six* words SFD produces. *Precision*, in information retrieval, is defined as the quantity of words in the query that exist in the corpus, divided by the cardinality of the query. That is, the precision is calculated as the percentage of terms from the fault diagnosis that were successfully found in the bug report.

Precision: Precision (as defined by information retrieval) of the words selected from the source code when compared against a corpus from the bug report.

Also note that Experiment one evaluates precision, but not recall. Recall, in information retrieval, is defined as the quantity of terms in the corpus that are found in the query, divided by the quantity of terms found in the corpus. That is, the recall would be defined as the percentage of the words in the bug report text that were part of the fault diagnosis. Bug reports often contain large amounts of text, at least relative to the few terms (six) in the diagnoses, and most are merely required for their sentences to be linguistically correct. Prior experience with bug reports has demonstrated that they often contain long paragraphs that include discussions between developers and test-case inputs that can be used to recreate failures. As such, the recall scores would not be a metric for which SFD would want to optimize. Instead, I consider that determining if the technique's produced terms were part of the discussion (*i.e.,* precision) as the meaningful metric.

Consider that this evaluation supposes that automatic diagnoses could have been generated *before* the bug reports were written. In such a simulation, correctly "predicted" words that would later be written by actual developers would constitute a success. I recognize that bug-report text is not a perfect oracle for evaluating the words extracted from the program: many words are used in the bug-report comments that may not be relevant or may use different, but synonymous, word choices. To increase the likelihood that words chosen by SFD match semantically meaningful words from the bug reports, I remove all stop words prior to comparison. If the terms that SFD automatically identifies as most suspicious are contained within the bug reports, I consider these diagnoses as likely successes in the production of relevant, useful, and meaningful words — that is, the automatically generated list of words matched words chosen by actual developers.

As such, the initial steps for this experiment are to create the selective corpora that represent the bug report as described in the independent variable I2. Then, to perform SFD and calculate the results for the bug listed in the bug report as described in the independent variable I1. After the various selective corpora from the bug report are generated and the

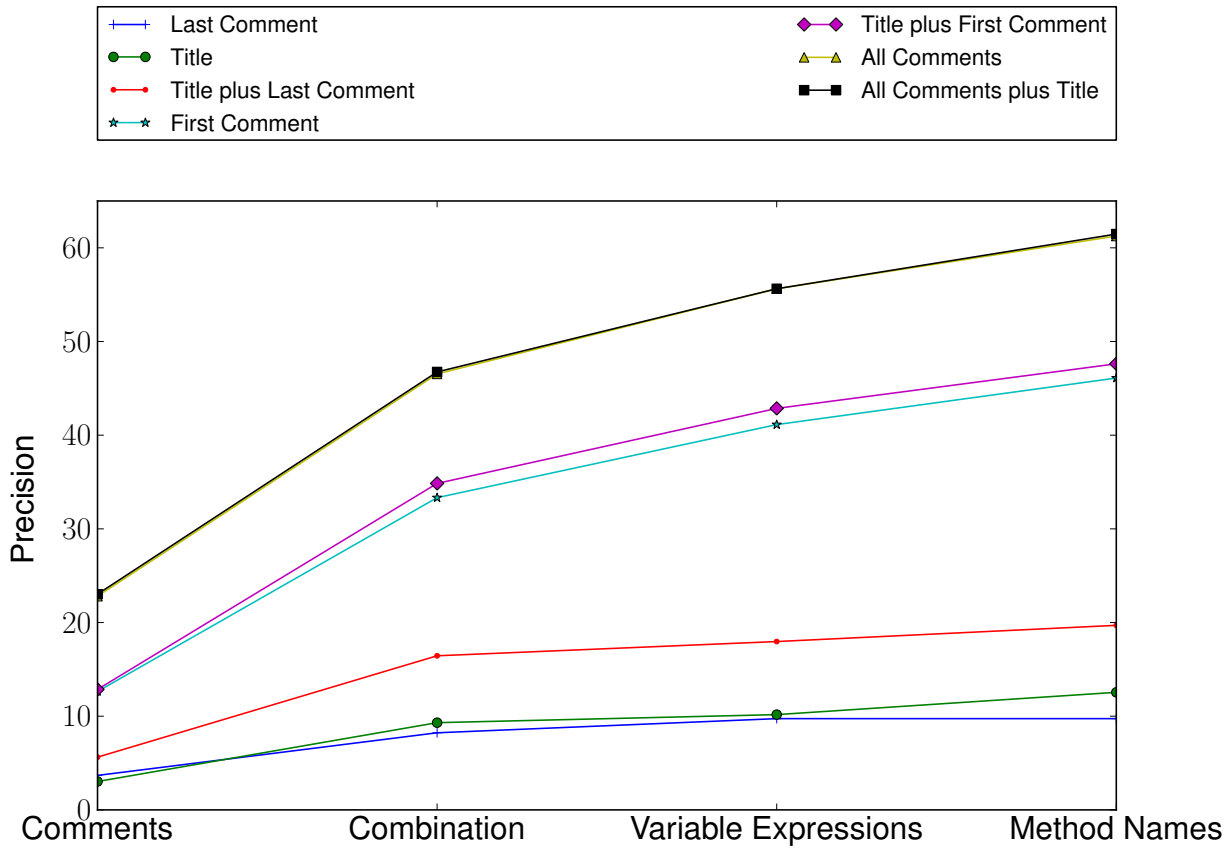

Figure 7.1: The precision of fault diagnosis results from MANTIS compared with bug report text from ASPECTJ.

source code is mined as described, the next step is to evaluate the four fault diagnoses. This last step allows for analysis about which parts of the source code were the most relevant to the diagnosis's success, and also which parts of the bug report was matched by the fault description. This is then repeated for each faulty version for all the 75 ASPECTJ versions.

7.2 Results

The results of Experiment one are presented in Figure 7.1. The four fault diagnoses are presented in columns along the horizontal axis. The mean precision score across all faulty ver-

sions varies along the vertical axis. These results show that the method-name and variable-expression categories outperform the combination and comments categories in their ability to generate the same words used in the bug reports. Unsurprisingly the larger subsets of text taken from the bug report result in higher precision scores.

When considering all post contents and the title, and using either the variable expressions or method names, more than half of the automatically produced terms could be found in the bug reports — on average, 55% of the terms from the variable-expressions category and 61% of the terms from the method-names category were found in the bug reports. The precision results from the comments category were worse: when comparing against all post text and title, only 23% of the terms from the comments category were found.

7.3 Analysis

In order to assert validity/invalidity of hypothesis *H1*, and to help interpret what a precision score of 61% means, I reference studies done by Furnas *et al.* and Copek and Szpakowicz [13, 28]. These two studies investigate the difficulty that humans have when attempting to describe behavior of a system using the same language. In other words, these two studies investigate how likely it is that two people will describe the feature of a system using the exact same language (*i.e.,* a precision score not accounting for synonyms or related words). Both of these studies found that this is a very difficult task, with Furnas finding that given six words (the same quantity used in this study), humans scored between 48 and 60% while Copek found that given ten words (more than used in our study) scores averaged at 28%. Thus, receiving a score of 61% means that Mantis is scoring similarly (or better) to a real human performing the same task given the same quantity of words.

An additional more thorough investigation of the results to ascertain why certain categories

of results were more successful than others (*i.e.,* method names had a higher precision than comments) was also performed. The comments category performed with relatively low precision scores in Experiment one, however, I found that the comments were useful for truly indicating the nature or feature domain of the fault. The primary reason comments had a lower precision is that often the identifiers and word choices in the comments did not exactly match the terms found in the bug reports; in other words, they scored poorly because they were synonyms. A human interpreting the comments would likely gain understanding from the comments. Unfortunately, the experimental design of comparing exact strings with the bug report text does not allow for demonstrating such benefits. The same can be said for many of the non-matched words from the variable-expressions and method-names categories. Whereas those categories matched 55–61%, on average, the remaining 39–45% of the terms were often relevant to the description of the fault. These were often interpretable for a human, but not found in the bug reports. I found that (in part) what allowed the method names category to have a high precision score is that bug reports often contained a stack trace, or method indicative to the functionality causing failure. These types of suspicious methods are most often what MANTIS returned in this category.

For each category, I also analyzed the words that were likely incorrect, meaning they were not found in the bug report and were not synonyms of bug report words. These words are selected primarily for two reasons: (1) the fault occurs in a portion of the code that is executed in each test case, or (2) the failure invokes functionality not found in many test cases, but not fault related. I found that the first case most often occurs because statistical fault localization also performs worse in these conditions, and statistical fault localization is used to correlate instructions with failure. I believe this shortcoming for MANTIS can be overcome by improving statistical fault localization to account for these types of faults or using another, additional form of fault localization to help inform likely faulty locations in the code. I found that in the second case, often those identified words were symptoms rather than causes of the faults, and hence there is a possibility of assisting in focusing developers'

attention.

As such, for hypothesis *H1*, I assess:

> As much as 61% of the automatically generated terms appeared in the human written bug reports, providing strong indication that such semantic fault-diagnosis techniques describe faults with similar key words to those written manually by real developers. The hypothesis was not invalidated.

7.4 Threats to Validity

While my results from Experiment one are encouraging, I recognize threats to their validity. Regarding construct validity, in Experiment 1, the use of bug report text as an oracle for the strings that *should* have been extracted is suspect. I recognize that in some cases of specific bug reports the text may include non-descriptive text or describe the wrong problem. However, because I studied (1) only bug reports that were fixed and reported the source code commit, and (2) a programming tool (AspectJ), whose users are likely to have more technical expertise, I expect the bug reports to be of high quality.

Another consideration with regard to construct validity involves the inability to truly measure "usefulness" or "meaningfulness" based on string matches. However, because all our studied faults were fixed, and real developers utilized the bug reports to find them, the choice to compare against these words seems a reasonable approximation.

One threat to the external validity is that each faulty version contained only a single fault. Thus, I cannot make absolute generalizations to systems that contain more than a single fault. However, as mentioned in Section 5, even if systems with multiple faults, a single fault can be evaluated by limiting the test suite and in cases where each test triggers multiple faults, existing studies suggest (and an anecdotal investigation on AspectJ verified) that one

fault is described with similar effectiveness [20, 21, 23]. The limitation however is that in this circumstance, developer has no control over the fault that is described with MANTIS. Thus, while more studies are needed to experimentally investigate this concern, existing data suggests that it is only a problem where developers needs to describe a specifically chosen, a priori, fault and have no test case that exercises only said fault.

I also recognize that MANTIS is dependent upon developers following reasonable naming conventions and including meaningful comments in their code. However, in those circumstances where developers name their variables with semantically-meaningful identifiers and make an effort to comment their code, such natural language extraction techniques can be beneficial.

Chapter 8

Hypothesis H2

H2: Automatically generated natural-language fault descriptions produce quality descriptions that accurately describe the fault.

- $H2_1$: *Does Semantic Fault Diagnoses contain words that are descriptive of the fault?*
- $H2_2$: *Is Semantic Fault Diagnoses helpful in understanding the fault?*

Because the purpose of semantic fault diagnosis is to assist human understanding, these hypotheses involve real developers through qualitative assessments of SFD results.

8.1 Experimental Design

The summarization of the experimental design, the hypotheses addressed, and the benefits and limitations are given in Table 8.1.

In order to evaluate the SFD approach, this experiment measures the *quality* of its output. In

Table 8.1: Summary of Experimental Design.

Hypotheses Addresssed	*H2:* Automatically generated natural-language fault descriptions produce quality descriptions that accurately describe the fault. $H2_1$: *Does Semantic Fault Diagnoses contain words that are descriptive of the fault?* $H2_2$: *Is Semantic Fault Diagnoses helpful in understanding the fault?*
Object of Analysis	Tetris (2+ KLOCs, 4 hand-seeded bugs)
Independent Variables	**I1.** Program Fault | SFD Results for different faults
Dependent Variable	**D1.** Qualitative assessment of SFD fault specificity
	D2. Qualitative assessment of SFD helpfulness
Benefits	**B1.** Verification that SFD words are fault specific **B2.** Assessment that SFD words are helpful understanding a fault **B3.** Evaluation by real developers using SFD results
Limitations	**L1.** Due to limited number of subjects and faults, reduced generalizabilty **L2.** Because every user is different, reduced repeatability

this context, I define "quality" as the ability to convey meaningful, fault specific information about a specific fault that is "helpful" for developers attempting to understand the fault. In this experiment, I assess the semantic value of the SFD output, presented as a Buggle (see Section 4), by performing a user study with five real developers, who were tasked with understanding the faults and providing subjective assessments of the quality of the SFD output.

I experimented using one *independent variable* — program faults. As the goal of SFD is to describe a fault, this experiment focuses on four different types of faults that either focus on a single functionality within the program, of span multiple different functionalities. Each fault was hand-seeded with the goal of affecting one of the major features of the game based upon the specification given by the creator.

Independent Variable 1: Program Fault.

In order to determine the quality of the SFD result, I focus on four hand-seeded faults that affect at least one major feature of the game Tetris:

bug 1. The functionality to drop blocks to lowest level moves them incorrectly.

bug 2. The key command to move blocks left instead moves them right.

bug 3. The pause functionality does not block user input (only game-automated events).

bug 4. The functionality to clear a full line only partially removes said line.

Dependent Variable: SFD quality. To assess the effectiveness of the SFD technique, I measured the quality of the SFD result (recall the previously stated definition of quality is that it is fault specific and helpful). This experiment measures semantic quality in two different ways:

Experiment 2 Metric: Fault Specificity. Developer subject assessed score, on a seven-point Likert scale, of the fault specificity of the diagnosis provided by MANTIS in the form of a Buggle, answering the question, "I think this picture describes/indicates the failure behavior."

Experiment 2 Metric: Helpfulness. Developer-subject assessed score, on a seven-point Likert scale, of the helpfulness of the diagnosis provided by MANTIS in the form of a Buggle, answering the question, "I think this picture would be helpful to debug the program."

In order to measure whether MANTIS results in quality, semantically meaningful fault descriptions, I utilize real developers in the following way. Five software developers were recruited from the University of California, Irvine, and hold a range of educational and industrial experience (one with Ph.D., two with Masters, and two with B.S.). As the goal is for them to assess MANTIS, it is important that the developers are familiar with the software upon which MANTIS is being leveraged (as to reduce bias that their interpretation is influenced by a lack of understanding of the underlying software). As such, this study uses the popular game, TETRIS.[1] In this Java implementation of TETRIS, I seeded four faults, into its 2.5 KLOC.

I provided each developer subject with screen shots that unambiguously depict the failure behavior of the TETRIS game. In addition, each subject was provided with the source code for the faulty program. The subjects were provided with unlimited time to examine the

[1] http://www.percederberg.net/games/tetris/tetris-1.2-src.zip

screen shots and the source code so to attempt to understand why the code fails and what would be necessary to fix it. So as to not bias or prime the subjects, I *did not* provide the subject with any natural-language description of the fault or failure — the subject determines his/her own interpretation using the screen shots and source code, alone. Thus, for these four bugs, the users are each respectively an expert as to the cause of failure and how to fix it.

One by one, each Buggle (see Section 4) for a particular fault is scored. Each Buggle corresponds to a specific variable (e.g., fault one or fault two). The subjects receive the Buggles in random order, meaning no two experts see the results in the same order. Further, the subjects are *blind*: they do not know which Buggle corresponds to which fault.

I then surveyed the subject to assess the semantic value of the SFD output in terms of quality and helpfulness to comprehend and fix the fault. The results are reported on a seven-point Likert scale, as described previously. Each subject repeats this process four times, each time for a different fault . Note that, for the duration of the study, I ask the developers to "think aloud" while I transcribe their comments, thus enabling a more explicit understanding of their choices.

8.2 Results

The results of Experiment 2 are presented in Figure 8.1. In Figure 8.1 I exhibit the mean[2] response score for quality of fault diagnosis for each fault. For each figure, the vertical axis represents the surveyed answers on a seven-point Likert scale, where 1 represents "strongly disagree," 2 represents "disagree," 3 represents "weakly disagree," 4 represents "neutral," and so on.

[2]Based upon subjects' responses and scoring patterns, these numbers appear to be interval, enabling parametric analysis.

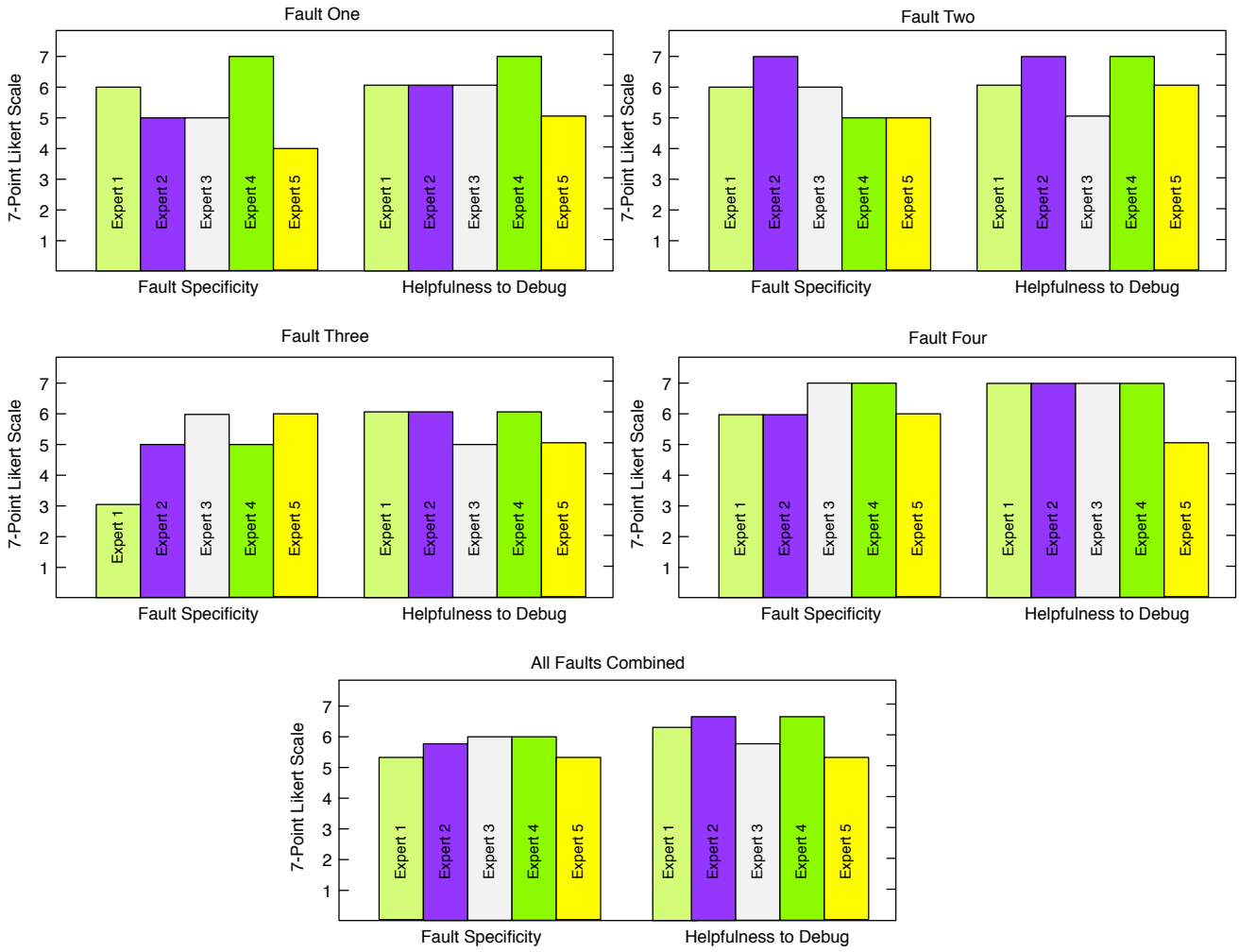

Figure 8.1: User study result assessing diagnosis quality separated by fault. Higher scores indicate high quality diagnoses.

The results are presented in groups for each independent variable (*i.e.,* each fault), and additionally a single graph representing the average for each expert when all faults are considered together.

Because the standard Likert scale typically produces ordinal numbers — numbers which relate based upon an ordering where the difference between disparate intervals may represent different quantities — the figure displaying overall results should be only used as a reference for ease of interpretation, and not a strict parametric analysis. More simply, because a Likert scale isn't discrete, the difference between one and two can be vastly different than the difference between four and five. Thus, taking the mean of those values can produce slightly misleading results. Therefore, it is only being used here to ease interpretation and all analysis will be based upon individual scoring.

The figure demonstrates that for these faults, each expert typically thought the descriptions were fault specific. In this case of the seven-point likert scale, a four is neutral, meaning all scores above such indicate that the description does signify the specific failure cause to some extent. The fault specificity received a positive (*i.e.,* greater than four) score for all faults and all experts except in two cases (one resulting in a neutral response, and another in a "slightly disagree" response).

Additionally, the helpfulness results in Figure 8.1 are almost uniformly, slightly higher in all categories than the fault specificity results. Indeed, most experts scored SFD results very positively when considering helpfulness.

It also should be noted that it appears that there is some difference in regards to the faults themselves. For example, fault three has slightly lower scores in both dependent variables than all other faults, while fault four scores the highest across both variables. While the amount of data is too small to perform a meaningful statistical analysis, it suggests that Mantis results are somewhat dependent upon the fault being described.

In addition to the quantitative Likert-scale surveyed responses, I also gathered qualitative think-aloud impressions of the subjects. When trends were observed in the qualitative results, I highlight some findings.

In regards to fault specificity, the experts typically felt that the MANTIS produced results that helped explain most of, but not all of the context. Indeed, comments like, "Its useful because at least I know it has something to do with <functionality>...but I think its missing some information about what happend" or, "its only got a little bit of noise, this is good" were not uncommon. Qualitatively, the subjects often commented that while a result was not completely accurate, it, "tells [them] where to start," and informs them of, "what [they] are looking for." Such comments explain how it is possible for a fault comprehension to be helpful even if the diagnoses are not perfect.

8.3 Analysis

Each research question is treated individually, and then collectively examined to answer hypothesis *H2*.

$H2_1$: Semantic Fault Diagnoses contain words that are descriptive of the fault.

In this study, the experts judged the specific fault relevance of the words provided by MANTIS. Their assessment in part, was to distinguish whether SFD provided results that in general described the program, or execution, or whether they provide semantically valuable, fault specific words. Quantitatively, their responses indicate that for any given expert, they believe that the words presented by SFD are indeed failure specific. This is indicated by the overall positivity of their scores in regards to this metric. Indeed, that SFD only had one negative response across all developers and faults is a highly encouraging signal that

SFD does specifically target and describe a specific fault. Further, this is reinforced by their statements regarding the results they analyzed. While it is clear that MANTIS results do not present the entirety of the context surrounding a fault, and to some extent have some noise, (as can be inferred by a lack of perfect scores and from their comments indicating this behavior) the consensus is that SFD results do indeed describe a specific fault and not general program behavior. In other words, we see observe positive scores given by actual fault experts that evaluated SFD results in terms of its ability to describe faults. However, as noted, because of the slight noise mentioned by a few experts, it is clear that there is room for MANTIS to become more precise and clear in its explanations.

As such for research question $H2_1$, I assess:

> MANTIS provided semantically valuable and meaningful diagnoses that specifically describe the fault, though not completely.

$H2_2$: Semantic Fault Diagnoses would be helpful in understanding the fault.

Just as fault specificity is a valuable signal to approximate quality (the main goal of hypothesis *H2*), another approximation of this is helpfulness. As the goal of MANTIS is to enable developers to better understand (and thus fix) their faults, ascertaining a measure of how helpful it would be for that task is an important measure of its quality. As such, each expert described whether they believed that MANTIS results would assist developers who had no understanding of the fault. These results indicate that our experts strongly believe that these MANTIS would be helpful. Quantitatively, for any given fault, the majority of experts score SFD with at least a six (*e.g.,* "strongly agree"). Qualitatively, as mentioned, the experts described that while the explanations given by MANTIS were not perfect, they clearly indicated a feature that was suspect, and provided strong clues about where to start

looking, and what was being looked for.

As such for research question $H2_2$, I assess:

MANTIS provided results that would be helpful in actually understanding the fault.

Overall Interpretation of Results

Hypothesis *H2* is designed to target a specific component of MANTIS, primarily, are the results of quality. While quality is an abstract idea at best, and contested idea at worst, I have attempted to approximate it with two measures. Each of these measures points to the heart of what MANTIS is trying to accomplish. Indeed, if MANTIS described bugs accurately, and is also helpful, its primary purpose would be fulfilled.

In this study, the experts for these faults found that overall, they believe MANTIS to produce quality results. While there is definite agreement that SFD can do more to eliminate noise and provide a more rich contextual understanding, these initial findings are promising. This study demonstrates that for these bugs, while MANTIS has room for growth, in its current state it is providing accurate, fault specific key words that would be helpful to a developer actually trying to debug.

As such, for hypothesis *H2*, I assess:

Semantic Fault Diagnosis can produce words that experts find accurate, descriptive of the fault, and helpful to developers. The hypothesis is not invalidated.

8.4 Threats to Validity

While the results for this study are positive, the threats to the validity of this experiment and these conclusions cannot be ignored. The single largest threat is to external validity. It is of note that this study only examined a single program, with four bugs, across five developers. As such, it is unclear whether these results translate to other programs or faults, and whether all developers would feel this way. However, the developers selected had fairly wide difference in their experience background (ranging from over two decades with solid industry experience to a grad student who has only ever coded in an academic environment). As such, because the developers had such different background and yet still were mostly united in their responses, it suggests that these results might generalize to some extent.

Another concern is with construct validity, in that instead of actual faults encountered during development, hand-seeded faults were used. As such, there is no clear indicator that these hand-seeded faults are representative of real faults, and that the results for real faults would be similar. That being said, the mutants were seeded specifically targeting functionality that was major within the program, meaning they represented problems with the primary functionality of the game. As such, they are to some extent representative of broken functionality that exists across many software platforms. Finally, past studies by Offutt *et al.* and Andrews *et al.* have demonstrated that mutants typically do not significantly alter the test results [4, 55, 56] for the evaluation of automated debugging techniques.

Chapter 9

Hypothesis H3

H3: Semantic Fault Diagnoses reduce the cognitive burden when attempting to understand a fault more than existing location-based debugging techniques.

- $H3_1$: *Is it less time consuming to interpret a Semantic Fault Diagnosis than a location based fault description?*
- $H3_2$: *Do developers gain a clearer understanding of a fault from a Semantic Fault Diagnosis than from a location based fault description?*
- $H3_3$: *Do developers find Semantic Fault Diagnoses more "helpful" to understand faults than a location based fault description?*

9.1 Experimental Design

The summarization of the experimental design, the hypotheses addressed, and the benefits and limitations are given in Table 9.1.

Table 9.1: Summary of Experimental Design.

Hypotheses Addresssed	*H3:* Semantic Fault Diagnoses reduce the cognitive burden when attempting to understand a fault more than existing location-based debugging techniques. $H3_1$: *Is it less time consuming to interpret a Semantic Fault Diagnosis than a location based fault description?* $H3_2$: *Do developers gain a clearer understanding of a fault from a Semantic Fault Diagnosis than from a location based fault description?* $H3_3$: *Do developers find Semantic Fault Diagnoses more "helpful" to understand faults than a location based fault description?*	
Object of Analysis	Tetris (2+ KLOCs, 4 hand-seeded bugs)	
Independent Variables	**I1.** SFD technique	$i1_a$. Mantis
	I2. Statistical Fault Localization technique	$i2_a$. Tarantula
Dependent Variable	**D1.** Qualitative assessment	Blind assessment of developer interpretation of results.
	D2. Questionnaire and interview	Developer assessment of helpfulness
	D3. Time	Time to complete investigation and interpretation of results.
Benefits	**B1.** Evaluation of the semantic quality of SFD words **B2.** Verification that SFD words are fault specific **B3.** Evaluation by real developer using SFD results	
Limitations	**L1.** Due to limited number of subjects and faults, reduced generalizabilty **L2.** Because every user is different, reduced repeatability	

Experiment Three involves the actual study of human developers using the results of the tool; it enables the witnessing of real use, along with identifying the strengths and limitations from practical experimentation. To assess the comparative *usefulness* of Mantis results to human developers in relation to existing techniques, I designed and implemented Experiment Three.

This experiment leverages a software system, with source code, that can be understood by an experimental subject developer in a short period of time. I used software that subjects were already familiar with, whose functionality is already understood. Further, the experiment requires a set of faulty versions from a software system, and that each faulty version contains only a single fault.

Experiment Three uses the game Tetris[1] as the software subject. Tetris was chosen

[1]http://www.percederberg.net/games/tetris/tetris-1.2-src.zip

because most of the subjects were likely to have a general understanding of the game rules, how to play, and the expected behavior. This understanding of general behavior means that their assessment of the debugging aids isn't harmed by a lack of understanding of the system in general. The source code is about 2,500 lines of code, containing roughly 1,000 lines of comments, and is written in Java. I created four versions, each containing a unique, hand-seeded fault. The four faults versions are: (1) the "left" key incorrectly moves the falling block, (2) the "down" key incorrectly moves the falling block, (3) the user remains able to move the block when the game is paused, and (4) a completed row of blocks is not fully cleared. While these faults are hand seeded, past studies by Offutt *et al.* and Andrews *et al.* have demonstrated that mutants typically do not significantly alter the test results [4, 55, 56] for the evaluation of automated debugging techniques.

Independent Variable 1: SFD technique.

Experiment three has two independent variables, which correspond to the two automated debugging aids: the first is an SFD technique (*e.g.,* MANTIS).

Mantis: The results generated from the MANTIS system for a specified fault.

Independent Variable 2: Statistical Fault Localization technique.

Experiment three has two independent variables, which correspond to the two automated debugging aids: the second is an existing automated-debugging-assistance tool, like a statistical fault localization tool (SFL) (*e.g.,* TARANTULA).

Statistical Fault Localization: The results generated from the statistical fault localization tool for a specified fault, in this case, TARANTULA [39] with the Ochiai metric [2].

The presentation of results to users is somewhat dependent upon the debugging assistance tool used, meaning their might be variance in other SFL techniques were used in place of

Tarantula, but the intention is to provide sufficient data such that a developer could attempt to understand and describe a fault.

Dependent Variables.

Experiment three has three dependent variables, highlights different types of assessment from real developers as to the quality of the answers given by users of the respective tools.

Qualitative assessment: A blind, third-party assessment of the developer fault description as aided by one of the tools.

Questionnaire and interview: A user survey relating to helpfulness and value of the system in aiding their interpretation.

Time: Time to complete the investigation and formulate a description of the fault with the aid of one of the tools.

For reference, when given Tarantula results, the subjects were also given the entire source code, and the links to the all lines with a suspicious score above 0.90 for ease of navigation. In our experiments this resulted in roughly 34–40 lines depending upon the faulty version. In contrast, the SFD results were given without source code: developers were just provided the top-10 suspicious terms from each category in Mantis (this experiment uses variable expressions, method names, and comments as its categories), along with their suspiciousness values for each term.

The human subjects for this study consisted of twelve computer science students at the University of California, Irvine. The subjects were chosen at random: they were approached in the hallways and classrooms in the computer science buildings. None of the subjects were affiliated with the author: they were not students in his courses nor affiliated in his research group. I required that each subject have at least one year of programming experience with Java with the vast majority having over seven years experience. The subjects ranged from second year undergraduates to fourth year graduate students in the computer science

department.

Each user completes two tasks in two phases (one in each phase), in the first phase the user is given a fault and a single assistance technique (either TARANTULA or MANTIS) and asked to provide a description. In phase two, the user is then given whichever assistance technique they did not utilize in phase one, and asked to augment their description leveraging this new information. Thus, this study has each user perform one pair of tasks:

Task 1a: using TARANTULA and the source code, provide a description for a given fault.

Task 1b: use MANTIS to try and augment the description generated in Task 1.

Task 2a: using MANTIS, provide a description for a given fault.

Task 2b: use TARANTULA and the source code to try and augment the description generated in Task 3.

Once a user has finished Task 1a (or Task 2a), they must be immediately given Task 1b (or Task 2b) so that the results are fresh in their mind. For each task, the user is given the appropriate artifacts (*e.g.,* MANTIS or TARANTULA with the source code) and asked to determine the cause of failure and provide a description of the fault. At the end of each task, the user is asked to give a free-form, spoken explanation of what they believe the fault to be that caused testing failure. The subjects' explanations and responses are recorded and then later transcribed. After both tasks, a user is asked to specify which of the two techniques they thought was more helpful, and give an explanation for their choice.

Note that each subject was given roughly 10 minutes of training on both MANTIS and TARANTULA (individually) and were not permitted to attempt any tasks until they could correctly answer questions by the researchers to ensure a sufficient understanding of how to use each technique. The users were be given the same amount of time to complete each of the two tasks, five minutes for each task. While the total allowable time is the same for each task, the time actually taken to complete the task was recorded. The task is considered

"complete" when the user is done examining the artifacts and announces that they are ready to provide a fault description. The time taken to provide the description was not be recorded; the user had all the time they need to relay their description (though once a user starts giving a description of the fault, they no longer had access to any of the artifacts that informed their description).

The assignment of faulty version to the task is randomized so that each faulty version is equally likely to be chosen for each task, however any single subject is assigned the same faulty version for Task 1a (or Task 2a) as they are for Task 1b (or Task 2b).

To evaluate the quality of the subjects' fault explanations, a *blind* comparison of results is conducted. Three independent "judges", who were not involved in the experiment, were given the transcribed explanations of the fault without any label that identifies which information was given to the subject. In other words, the judges did not know which technique informed each explanation — they were not be able to differentiate an explanation from, say Tarantula, or Mantis. When the transcription contained phrases that would reveal the technique (*e.g.,* "this code is 95% suspicious, so I think that the bug is..."), the transcription was minimally redacted to preserve the judges' inability to distinguish them, so as to minimize the possibility of bias. This is one reason it was important to record a verbal response and not have the users write a response, as it in many cases, minimal redaction is necessary to ensure the judges remain *blind*.

The judges made three types of comparisons: (1) inter-user results from Task 1a and Task 2a (to determine the usefulness of each artifact in creating a fault description), (2) intra-user results from Task 1a and Task 1b (to determine the usefulness of being given Mantis after having access to Tarantula and the source), and (3) intra-user results from Task 2a and Task 2b (to determine the usefulness of being given Tarantula and the source code after having access to Mantis). The judges were instructed to choose between a given two fault explanations on the basis of which more accurately and completely described the fault. In

the case where a the descriptions were sufficiently similar, or user could not augment their description after being given the additional artifact such that their description is identical, the judge is informed to mark them a tie.

Note that when selecting judges, I selected only developers with at least ten years of java experience and at least four years industry experience. This was done to ensure that all the judges were accomplished coders, familiar with debugging and had seen a variety of bug reports or descriptions from fellow employees. To ensure that each judge had a proper understanding of the fault they were provided with the full description of what was faulty in the program for each of the faults. It is important to note that when providing this *description* of the fault to the blind judges, care was taken as not to bias them toward a particular technique. This was done by allowing them to see the fault in action, examine the source code, the test cases, and debug the fault themselves so they have a working knowledge thereof. As such, each judge was given the faulty version and a test suite with at least one passing and one failing test cases (note that there were 20 test cases total) to enable exploration of the fault. The judges were then asked to describe their understanding of the failure to the researchers to ensure that there was no misunderstanding in the fault for a given version.

Upon completion of the judging, the results were tabulated to reveal the performance of the techniques. To do so, I "un-blinded" the judges' assessments, *i.e.,* I revealed which answer came from which technique, for each of the compared pairs. In this way, the results demonstrate which technique (in isolation) is more effective at enabling fault comprehension, and if combining these techniques results in an even better fault description than either technique in isolation.

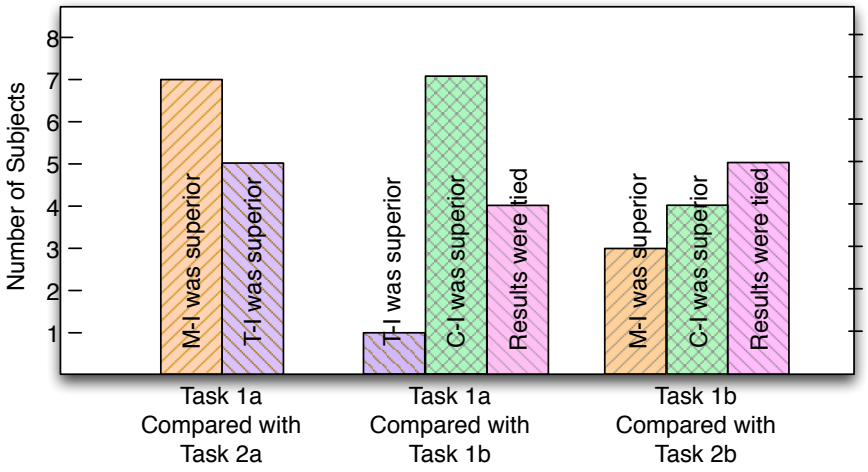

Figure 9.1: Quality results for Experiment three segregated by task. M-I means "Mantis informed decision", T-I means "Tarantula informed decision", and C-I means "Mantis and Tarantula informed decision".

9.2 Results

I first present the results for Experiment three, and then individually address the three research questions this experiment is designed to answer.

Figure 9.1 presents the results in three groups corresponding to the three types of result comparisons, inter-user Mantis against Tarantula, intra-user augmenting Tarantula with Mantis, and intra-user augmenting Mantis with Tarantula.

In the left pair of bars, the left bar represents the number of subjects for which the fault explanation informed by Mantis alone was better than the explanation informed by Tarantula and access to the source code. The right bar presents the number of subject's descriptions that were judged as better with Tarantula and the source code. For seven comparisons, the subjects provided a better explanation with only the Mantis result and no source code. For five comparisons, the subjects provided a better explanation with the source code and the Tarantula result.

In the middle group of bars, the left bar represents the number of subjects for which the

fault explanation informed by Tarantula was judged as better than the explanation informed by Tarantula and Mantis. The middle bar represents the number of subject's descriptions that were judged better when informed by Tarantula and Mantis rather than Tarantula alone. In other words, the middle bar represents subjects who started with Tarantula and then augmented their understanding with Mantis and had a more complete answer. The right bar represents the number of subjects that choose not to, or could not augment their description after gaining access to Mantis results. One comparison showed a developer performing better with Tarantula alone, meaning that in that case, the developer's description degraded after seeing Mantis results. For seven comparisons, the explanation improved after gaining access to Mantis results, and for four comparisons had developers could not to augment their description utilizing Mantis descriptions.

In the right group of bars, the left bar represents the number of subjects for which the fault explanation informed by Mantis was judged as better than the explanation informed by Tarantula and Mantis. The middle bar represents the number of subject's descriptions that were judged better when informed by Mantis and Tarantula rather than Mantis alone. In other words, the middle bar represents subjects who started with Mantis and then augmented their understanding with Tarantula and source code to have a more complete answer. The right bar represents the number of subjects that choose not to, or could not augment their description after gaining access to Tarantula results and the source code. Three comparisons show that developers were better with Mantis alone, meaning that their description degraded after seeing Tarantula and the source code. Four comparisons show that developers improve their description after gaining access to Tarantula and the source code. Finally, five comparisons had developers feel as if they could not augment their description utilizing Tarantula and the source code.

Throughout the experiment, the time required for each task was recorded. Additionally, at the conclusion of all four tasks, each subject was asked which form of result they felt

was more helpful. Table 9.2 presents these assessments of relative helpfulness of the two techniques' results, and the mean time required by the subjects to understand the results and give their explanations.

Eleven out of the twelve subjects thought that the MANTIS results were more helpful than the TARANTULA results, one subject thought that they were equally helpful, and zero subjects thought that TARANTULA was more helpful.

Qualitatively, when discussing MANTIS, sentiments are summarized with quotes such as "[now] I know where to start looking" and "Ah. I understand the problem" and "these words suggest the problem is with [some specific] functionality". Generally, the user expressed that MANTIS results directed them toward a functionality or specific behavior, while the comments specified what that functionality was expected to do; enabling to know fairly clearly what they were looking for, and what to verify.

Whereas when discussing TARANTULA, sentiments are summarized with quotes such as "[I don't] undertand what this suspicious code is doing" and "these [lines] all have the same [suspiciousness] numbers, so, I'm not sure" and "I guess it must be these lines, but I dont see a legitamate error". Typically, our users utilized TARANTULA to focus on a specific subset of the code, but often were unclear why they were directed there; resulting in users who were often looking at areas of the code with the fault but not knowing if it was faulty.

Lastly, when augmenting their responses, sentiments are summarized with quotes such as "having seen the code already, these words reinforce my thoughts" and "I looked for areas of the code with [many] of the words [from MANTIS]". Here users presented an interesting pattern of behavior. Users would use MANTIS to discover a type of behavior they were looking for, and then user TARANTULA to find the code that dealt with that behavior. In reverse, the users would use MANTIS to help *explain* to them why they were looking at a set of suspicious code from TARANTULA to help guide them in their search.

Table 9.2: Subjective assessment of helpfulness, and the mean time that the subjects required for task completion.

	Mantis	**Tarantula**
Technique more helpful	11 of 12 subjects	0 subjects
Mean time required for task completion	65 seconds	143 seconds

Regarding the time to task completion, the subjects were able to offer an explanation of the fault using the MANTIS results in less than half the time required when using TARANTULA.

9.3 Analysis

Each research question is treated individually, and then collectively examined to answer hypothesis *H3*.

$H3_1$: It is less time consuming to interpret a Semantic Fault Diagnosis than a location based fault description.

One successful value identified in this experiment is its ease of interpretation. Developers commented that by looking at the results, they almost immediately had an intuition of the problem, whereas with more complicated SFL results and source code, they had to invest effort to understanding surrounding code contexts, and attempt to ascertain likely control flow paths.

Due to the simplicity of the word list, the subjects quickly built an understanding of the fault nature. The fault localization results were more complicated and required reading source code, and so, required more time to interpret. While this follows intuition, I expect this difference to increase dramatically in larger or more complex systems.

As a further point on the issue of timing, the semantic fault diagnosis technique requires,

on average, 10 milliseconds to compute once the source code and test information have been loaded into memory on a Pentium i5 processor. I consider the technique to be quite scalable, and any I/O expense can be minimized or eliminated if the source code could be cached in memory on each commit or were already in memory.

As such for research question $H3_1$, I assess:

> Using the semantic fault diagnosis, the subjects gave their interpretation of the results in less than half of the time that they required for the fault-localization result.

$H3_2$: Developers gain a clearer understanding of a fault from a Semantic Fault Diagnosis than a location based fault description.

As displayed in Figure 9.1, an inter-user comparison of results between subjects given Mantis and those given Tarantula found that more subjects were able to come to an accurate conclusion as to the nature of the bug when using Mantis. It is of note that in those cases where Mantis informed descriptions were superior, the users had not seen any source code. This means that a user was able to provide a more accurate description of the fault only seeing a few keywords and comments extracted from the source than a user given the source and the failure correlated lines. I see this as a promising result that a meaningful description can be interpreted and enable sufficient understanding to ameliorate debugging.

The result from this inter-user comparison is reinforced with the qualitative responses previously mentioned as most Mantis users expressed ease and comfortability in interpreting the diagnoses, while many Tarantula users found "jumping" from code location to code location without an understanding of why to be confusing.

I see further divergence when we examine the results from the *augmenting* tasks. When a

user was given Tarantula first and then given Mantis after creating a description, in most cases their understanding improved, though there are a non-trivial amount of cases where their understanding remained the same. Lastly, only in one case did a user become confused and degrade in quality after seeing Mantis scores, suggesting that, if these results generalize, most users will stand to benefit from seeing Mantis results even if they have some existing understanding of the fault.

In contrast, when a user was given Mantis first and then given Tarantula and the source code, we see that in most cases their understanding remained unchanged, though there is a non-trivial amount of cases where their understanding did improve. Further, in 25% of the cases, users understanding degraded as a result of seeing Tarantula and the source code, meaning that if this experiment generalizes, utilizing Tarantula after coming to an understanding of the fault entails a non trivial risk of confusing the developer.

This difference in results and opinions suggests that not all debugging aids are equals in terms of understanding faults. Further, our judges noted that while the descriptions provided with the aid of Mantis were at a high level of abstraction (*i.e.,* primarily focusing on functionality) they were remarkably accurate (*e.g.,* the move left function moves the block in the wrong way). In contrast, the judges found for Tarantula aided descriptions, users tended to describe the fault at a very low level (*e.g.,* a variable initialized in this method is off by one) and either directly pointed out the fault, or were completely off.

For these results, I also performed a deeper investigation to understand when and why Mantis won and lost to SFL. It is of note that SFL performed quite well on this subject, localizing every single bug in less than 10% of the code. I found that when the Tarantula-informed description was more accurate, it was typically because SFL was able to isolate the fault to an individual method, and very little else. This allowed developers to rule out most of the code base, and understand where the broken functionality fits into expected control flow. However, when the Mantis-informed description was more accurate, the SFD results

typically described not just the fault, but the functionality involved with it, which allowed the subjects to better describe the problem and focus their attention.

As such, for research question $H3_2$, I assess:

> A majority of human subjects were able to describe the fault as well or better with only the automatically generated fault diagnoses than those that had the full source code and statistical fault localization support.

$H3_3$: Semantic Fault Diagnoses are more "helpful" to understand faults than a location based fault description.

Overwhelmingly, the subjects thought that the word lists provided by the semantic fault diagnosis tool were more helpful than source code and locational information. The only subject who did not state that he felt that the semantic fault diagnosis was more helpful instead stated that he felt they were equally helpful.

This is somewhat surprising in that not one subject selected Tarantula and source code as more helpful. Our interviews revealed that users were able to grasp the behavior Mantis suggested very quickly and easily. In other words, the users found that with a basic understanding of the program, which they all had for Tetris, they could as one user put it, "create a mental model" of the expected behavior that may be problematic by looking at the Mantis results.

In contrast, when discussing Tarantula, users felt like it was difficult for them to use as it was directing them to multiple non-contiguous locations in the code without any explanation. Further, while the users were familiar with Tetris's functionality, they were unfamiliar with the specifics from the code (*e.g.,* variable names and uses or locations and purposes for specific loops), which was "time consuming" and "difficult" to contextualize according

to the users. I speculate that this is because looking for a specific bug among nearly 40 lines of disjoint code meant multiple context switches and attempting to understand each of these lines' dependencies and place within a global control flow is a non trivial task. In contrast, MANTIS described the functionality involved in failure such that the users were able to abstract away these details and focus on the fault.

As such for research question $H3_3$, I assess:

> Eleven out of twelve subjects stated that they found the semantic fault diagnosis more helpful than the source code and fault localization results, clearly implying potential for improved early triaging and impact in practice.

Overall Interpretation of Results

As a whole, the results from Experiment 3 show that semantic fault diagnosis techniques can provide a faster, easier to interpret result that is more helpful and in general, enables a clearer explanation of the fault than existing fault locational based techniques. Further, for this experiment I found that SFD is efficient, both in terms of the computational time and the human interpretation time.

Additionally, despite the relative success of the semantic fault diagnosis versus the statistical fault localization, I do not foresee the former supplanting the latter. Instead, I foresee usage scenarios in which, first, semantic fault diagnoses offer early descriptions to developers about their faults, perhaps immediately after testing failure; and then once the developers are ready to start debugging those faults, using both the semantic results as well as the localization results to focus their attention in looking for the fault. This is reinforced by the contextual level of abstraction that each technique provided to developers (MANTIS enabled a clear

understanding of fault functionality which could then be localized by TARANTULA) and that in a subset of cases, adding this additional information enabled a clearer understanding to developers. Indeed these results suggest that combining these techniques may allow the developers to perform best.

However, these results reinforce that if only a single technique can be leveraged, MANTIS enables a faster, more helpful explanation. Thus for Hypothesis *H3* I conclude:

> For these experimental subjects, semantic fault diagnosis: was faster; produced descriptions which were more meaningful and more quickly interpretable. The hypothesis is not invalided

9.4 Threats to Validity

While the results from this experiment are encouraging, I recognize threats to their validity. With regard to a threat to external validity, the user study included only twelve subjects, who were all computer science students. While I cannot ensure that these results will hold for software developers in general, the results demonstrate the approach *can* produce useful and descriptive results.

An additional threat to external validity is that the program subjects were using was small (2.5+ KLOC) and may not have many complexities that are found within larger programs. As such, I cannot make any generalization about how MANTIS would perform if given a larger or more complex program. However, this study does reinforce that in cases where a developer has a general understanding of the program functionalities, MANTIS *can* more reliably convey proper context about the bug than existing techniques.

I also recognize that MANTIS is dependent upon developers following reasonable naming conventions and including meaningful comments in their code. However, in those circum-

stances where developers name their variables with semantically-meaningful identifiers and make an effort to comment their code, such natural language extraction techniques can be beneficial.

Chapter 10

Hypothesis H4

H4: Optimizing any step of the natural-language pipeline or fault localization within Semantic Fault Diagnosis similarly improves the results.

- $H4_1$: *Does optimizing text parsing provide higher quality Semantic Fault Diagnoses?*
- $H4_2$: *Does optimizing text normalization provide higher quality Semantic Fault Diagnoses?*
- $H4_3$: *Does optimizing dimensional-reduction techniques provide higher quality Semantic Fault Diagnoses?*
- $H4_4$: *Does optimizing the fault localization produce higher quality Semantic Fault Diagnoses?*

10.1 Experimental Design

The summarization of the experimental design, the hypotheses addressed, and the benefits and limitations are given in Table 10.1.

Table 10.1: Summary of Experimental Design.

Hypotheses Addresssed	*H4:* Optimizing any step of the natural-language pipeline or fault localization within Semantic Fault Diagnosis similarly optimizes the results. $H4_1$: *Does optimizing text parsing provide higher quality Semantic Fault Diagnoses?* $H4_2$: *Does optimizing text normalization provide higher quality Semantic Fault Diagnoses?* $H4_3$: *Does optimizing dimensional-reduction techniques provide higher quality Semantic Fault Diagnoses?* $H4_4$: *Does optimizing the fault localization produce higher quality Semantic Fault Diagnoses?*	
Object of Analysis	Tetris (2+ KLOCs, 4 hand-seeded bugs) and AspectJ (75+KLOCs, 75 bugs)	
Independent Variables	**I1.** Failure-Relevance	$i1_a$. True relevance $i1_b$. Random relevance
	I2. Splitter Technique	$i2_a$. Naive $i2_b$. Enhanced Camel Case $i2_c$. English Dictionary
	I3. Stemmer Technique	$i3_a$. K-Stem $i3_b$. M-Stem $i3_c$. P-Stem
	I4. Aggregation Statistic	$i4_a$. Mean $i4_b$. Max $i2_a$. Mode
Dependent Variable	**D1.** Precision	Precision of diagnosis compared with text from fixed historical bug reports
	D2. Fault specificity	Qualitative assessment of SFD fault specificity
	D3. Helpfulness	Qualitative assessment of SFD helpfulness to debug
Benefits	**B1.** Evaluation of each component in SFD workflow **B2.** Measurable assessment of how each component affects end result **B3.** Qualitative and Quantitative results **B4.** A greater understanding of potential optimizations	
Limitations	**L1.** Due to limited number of subjects and bugs, reduced generalizability **L2.** Because every user is different, reduced repeatability	

In order to evaluate the SFD approach, and understand how each component affects the quality of the end results, experiment four leverages two sub-experiments, primarily an expansion of those utilized in experiment one and experiment two. In each sub-experiment,

I assess semantic change of the results in different ways, specifically those used by the earlier experiments. The first sub-experiment is designed to understand how changes in the SFD workflow impact the precision of its results, and the second sub-experiment is designed to understand how changes in the SFD workflow impact the quality of its results.

In the first sub-experiment, I conduct a simulation of performing SFD for historical bugs in a project's source-code repository and bug-tracking system. Like before, I analyze 75 real faults from the open-source project AspectJ. In the simulation, as previously discussed, I perform SFD as if it were automatically run at the moment of witnessing failures — prior to bug reporting, investigation, and debugging by developers. To assess the semantic value of the SFD output, I calculate the intersection of the top six-most, automatically generated terms that SFD provides with all the terms developers actually wrote inside the bug report in their investigation and resolution for that fault.

In the second sub-experiment, I assess the semantic value of the SFD output, presented as a Buggle (see Section 4), by performing a user study with five real developers, who were tasked with understanding the faults and providing subjective assessments of the meaningfulness of the SFD output. This is the same format used in experiment two. To reduce the individual developer's bias, I presented the developer subjects with Buggles of varying SFD configurations *blindly* (i.e., they were not informed of which configurations were which).

These two sub-experiments address different goals, as they target multiple different components of SFD's workflow. A change to the fault localization will change the entire way that Mantis determines what is or is not suspicious, the entire event being described will potentially change. Note that a perfect SFD tool would describe would be the cause of failure, (*i.e.,* the fault). A change in the natural language components can result in different components of text being returned, different types of words being returned, different amounts of text being returned or text from different locations in the program are being returned. In essence, the same event is being described, but in a different way, which may significantly

Table 10.2: Example uses of three splitter techniques.

String to Split	Naive	Enhanced Camel Case	Dictionary
accountInfo(x.getResults(i))	accountInfo	accountInfo(x	count
	x.getResults	accountInfo	account
	i	account	tin
		info	in
		x.getResults(i)	info
		x.getResults	get
		get	result
		...	...

impact the quality of the results. Thus, instead of leveraging a single evaluation technique, each sub-experiment focuses on a result type to determine the overall impact upon the final results.

For both sub-experiments I assess the sensitivity of the SFD output to the configuration options to answer the hypothesis *H4*. This experiment uses four *independent variables* — splitter techniques, stemmer techniques, aggregation statistics, and failure-relevance statistics — and evaluated their influence on the effectiveness of the SFD technique by using the dependent variables precision, fault specificity and helpfulness to debug. Each variable will be defined and motivated in turn.

Independent Variable 1: Splitter Technique. In order to determine the degree to which the SFD result is sensitive to the particular source-code splitter technique, I implemented three popular splitter techniques:

Splitter 1: Naive. Strings are split on standard delimiters, such as "_", "(", ")", and ";".

Splitter 2: Enhanced Camel Case. Strings are split using "camel case" capitalization rules, and then subsequently split using method call signatures.

Splitter 3: Dictionary. Strings are split using an English dictionary.[1]

In Table 10.2, I provide an example of the three splitters. The naive splitter splits on specific delimiters; in this case it splits terms on parentheses. The enhanced camel-case splitter splits

[1] http://www.outpost9.com/files/WordLists.html

Table 10.3: Example uses of three stemmer techniques.

Terms to Stem	Porter (Rule-based)	Krovetz (Morphological)	M-Stem (Hard-Coded)
Elemental	Element	Elemental	Element
Rating	Rat	Rate	Rate

on each method call, parameter use, and every camel-cased word. For example, the identifier "x.getResults(i)" is split first on the method call, to generate "x" and "getResults(i)". Then it is split again on the parameter use returning "getResults" and "i." Finally it splits the camel case "getResults" and produces "get" and "results." In each instance the enhanced camel-case splitter returns the full identifier along with the split parts. The traditional English dictionary splitter, scans through every possible substring of the identifier, and stores each substring that matches an entry in the English dictionary. With this splitter, correct words like "account" are identified, but so are many irrelevant words, like "tin," which is extracted from "accountinfo."

Independent Variable 2: Stemmer Technique. In order to determine the degree to which the SFD result is sensitive to the particular natural-langauge stemmer, I use three popular stemmer techniques:

Stemmer 1: P-Stem. The rule-based, Porter stemmer.[2]

Stemmer 2: K-Stem. The morphological, Krovetz stemmer.[3]

Stemmer 3: M-Stem. The hard-coding-based stemmer, M-Stem.[4]

In Table 10.3, I provide an example of the three stemmers. P-stem follows encoded rules to remove the endings of words, thus removing the "al" from "elemental" (returning the correct stem), and the "ing" from "rating" (returning the incorrect stem). K-stem uses an algorithm to recognize the correct part of speech, and because it is unable to find the true root of "elemental," the result remains unchanged. K-stem removes the "ing" from "rating"

[2]http://tartarus.org/martin/PorterStemmer
[3]http://lucene.apache.org/solr
[4]`http://msuweb.montclair.edu/~hillem/stemming`

Table 10.4: Example uses of three failure-correlation aggregation techniques.

Scores to Aggregate	Mean	Max	Mode
11, 12, 21, 25, 47, 92, 95, 96, 97, 99	59.5	99	95.5

and inserts an "e", thus returning the correct stem. M-stem, has both words hard coded into its dictionary and thus produces the correct stems for both.

Independent Variable 3: Aggregation Statistic. In order to determine the degree to which the SFD result is sensitive to the aggregation statistic that is used to aggregate the failure-correlation scores of for each term, I implemented three standard aggregation statistics:

Aggregation Statistic 1: Mean. Each term is assigned the mean of all the failure-relevance scores across all instances of that term throughout the source code.

Aggregation Statistic 2: Max. Each term is assigned the max of all the failure-relevance scores across all instances of that term throughout the source code.

Aggregation Statistic 3: Mode. Each term is assigned the interval-based mode of all failure-relevance scores across all instances of that term throughout the source code.

I implemented the mean statistic to give an indication of whether the term most frequently occurred in failure-correlated code, but infrequently in test-passing-correlated code. I implemented the max statistic to give an indication of the terms that occur in the most failure-correlated code, regardless of where it appears in the rest of the codebase. I implemented an interval-based mode statistic to give an indication of the terms that most often occur in failure-correlated code and are not as susceptible to outliers as the other two statistics. As is standard for continuous data, the mode statistic is calculated by stratifying the scores for each term into discrete intervals (we used intervals of 10% correlation with failure), and then taking the average failure-correlation for the most represented interval. Table 10.4 presents an example of using the three statistics to aggregate a set of failure-correlation scores for a particular term.

Independent Variable 4: Failure-Relevance Statistic. To determine whether the degree to which SFD results are impacted by the dynamic information from fault localization I implemented two failure-relevance statistics:

Relevance Statistic 1: True. Each line of code is assigned a score by the degree to which its execution correlates with test failure using the standard Ochiai correlation metric.

Relevance Statistic 2: Random. Each line of code is assigned a random failure-correlation score, so as to identify common terms in the code.

Dependent Variables: precision and quality. To assess the effectiveness of the SFD technique, I measured two types of semantic value of the SFD result, precision and quality.

Sub-Experiment 1 Metric: Precision. The percentage of the top-six terms prescribed by MANTIS that occur in the text of the bug-report. MANTIS terms are mined from the source code (e.g., variable expressions, method names, developer comments) after they are ranked, stratified, and sorted. Bug report text describes the investigation and correction of the fault by actual developers. A high score indicates that SFD could have provided useful clues to the fault's nature prior to fault investigation.

Sub-Experiment 2 Metric: Fault Specificity. Developer subject assessed score, on a seven-point Likert scale, of the quality of the diagnosis provided by MANTIS in the form of a Buggle, answering the question, "I think this picture describes/indicates the failure behavior."

Sub-Experiment 2 Metric: Helpfulness. Developer-subject assessed score, on a seven-point Likert scale, of the helpfulness of the diagnosis provided by MANTIS in the form of a Buggle, answering the question, "I think this picture would be helpful to debug the program."

To measure the impact each independent variable has on MANTIS, I utilize a benchmark version, which is defined as using Enhanced Camel Case Splitter, Porter Splitter, Mean Aggregation Statistic, and True Relevance Statistic. Each other configuration of MANTIS is produced by altering the benchmark configuration by exactly one parameter. As such, each other configuration is referred to by its change from the benchmark configuration.

As the specifics of how both these sub-experiments are setup has been previously discussed

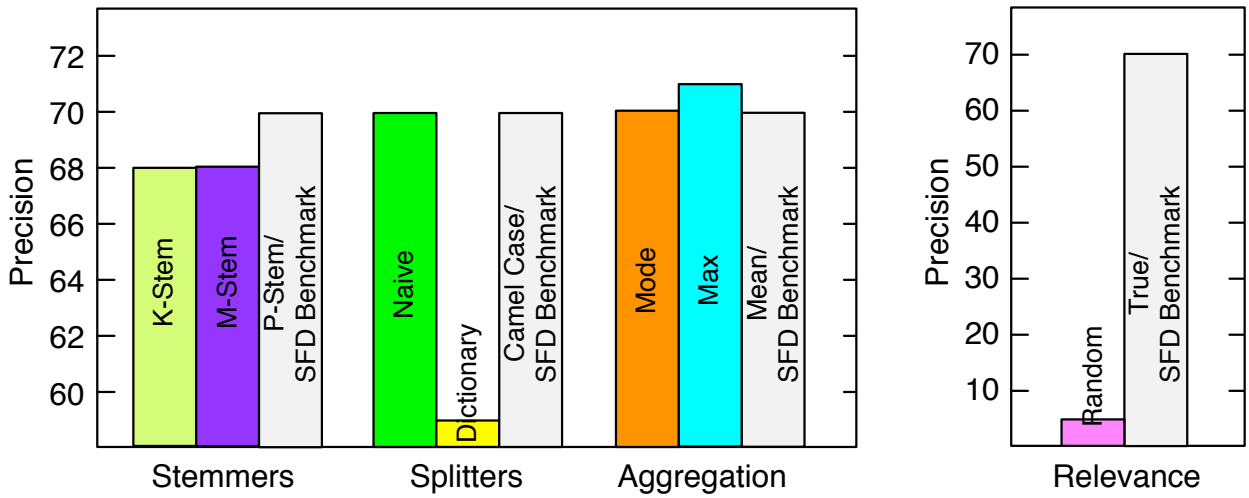

Figure 10.1: Experiment 1 results as the percent of SFD terms found in the bug-report text. Note the *difference in scales* for the two charts, the scale on the left ranges from 60 to 72, whereas the right ranges from 0 to 70.

(I refer the reader to Section 7.1 for the AspectJ study, and Section 8 for the user study) I will not repeat their setup here. However, the significant difference in these sub-experiments, are the types of Mantis that are used to create results. Whereas previously only a single configuration of Mantis was leveraged, now the same study is repeated eight times, once for each independent variable configuration.

10.2 Results

Each sub experiment will have its results treated separately.

Sub-Experiment 1 Results: The results for this evaluation are presented in Figure 10.1. The figure displays the effectiveness of each configuration, including the benchmark version of Mantis, for each independent variable value. The scores are given as percentages, where the left figure's vertical axis ranges from 60–72 and the right's vertical axis ranges from 0–70 in order to highlight the differences in scores.

Generally, the precision varies by only small amounts in the various configurations (with the exception of the Dictionary Splitter and Random). For a given configuration, there is a clear average among the techniques — the scores remain around 68–70%. These high precision scores are in contrast with the random relevance statistic, which demonstrates an order of magnitude difference with the true relevance statistic used in the benchmark configuration of MANTIS. This strong difference indicates that the SFD results are truly fault-specific, as opposed to simply extracting terms from the program that are program-domain-specific. As an aside, the average time taken (in seconds) to generate MANTIS results for each parameter are as follows: K-stem 18.21, M-stem 17.13, benchmark (P-stem/camel-cased/mean/true) 17.04, naive 7.56, dictionary 53.39, max 18.43, mode 6.18, and random 8.52.

I performed paired statistical t-tests for each set variable type. When considering splitters there was no statistically significant difference between the camel-case and naive (a p-value of 0.59), but there was a statistically significant difference between either of these and Dictionary (a p-value ¡ 0.001). When considering stemmers there was a small, but statistically significant difference between P-Stem and K-Stem (a p-value of 0.004); a small, but statistically significant difference between P-Stem and M-Stem (a p-value of 0.02); and no statistically significant difference between K-stem and M-stem (a p-value of 0.73); Finally when considering aggregation techniques, there was no statistically significant difference between mode, mean or max (p-values are all above 0.75).

Sub-Experiment 2 Results: The results of Experiment 2 are presented in Figures 10.2 and 10.3. In Figure 10.2 I exhibit the mean[5] response score for quality of fault diagnosis for each MANTIS configuration. Likewise, in Figure 10.3 I exhibit mean response score for helpfulness of the diagnosis for debugging. For both figures, the vertical axis represents the surveyed answers on a seven-point Likert scale, where 1 represents "strongly disagree," 2 represents "disagree," 3 represents "weakly disagree," 4 represents "neutral," and so on. Along the

[5]Based upon subjects' responses and scoring patterns, these numbers appear to be interval, enabling parametric analysis.

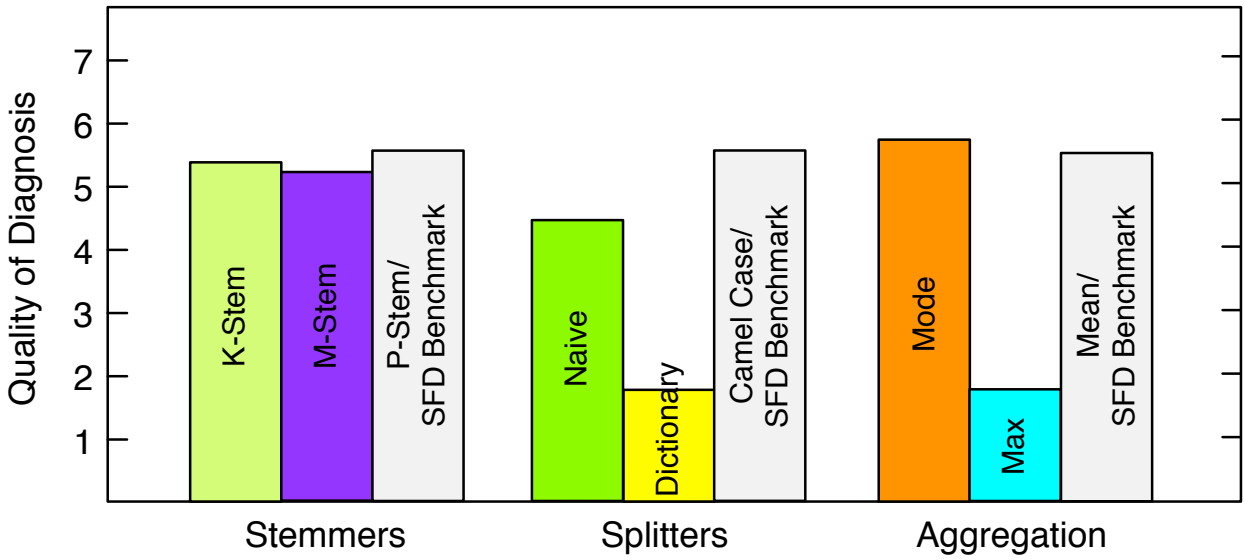

Figure 10.2: Experiment 2's user-study results assessing diagnosis quality. Higher scores indicate higher quality diagnoses.

horizontal axis we represent each of the different parameters, grouped by their type for ease of comparison. Note that in each grouping there exists the "baseline" version.

Like the results for sub-Experiment 1, the results are presented in groups for each independent variable, with the benchmark version of MANTIS in each category. Because the random failure-relevance statistic produced such poor results (5% precision), our user study only evaluates the first three independent variables.

While the standard Likert scale typically produces ordinal numbers due to the varying lengths of disparate intervals, based upon the experts responses it appears as if the intervals for this particular study, are fairly constant. I conclude this because of the verbal comments made during the study and the variety of scores that were chosen (each number including one and seven was used by every expert). It is likely that the results produce interval numbers rather than ordinal, enabling these graphs to utilize parametric analysis. Thus, each bar represents the mean of the configuration across all faults and all experts.

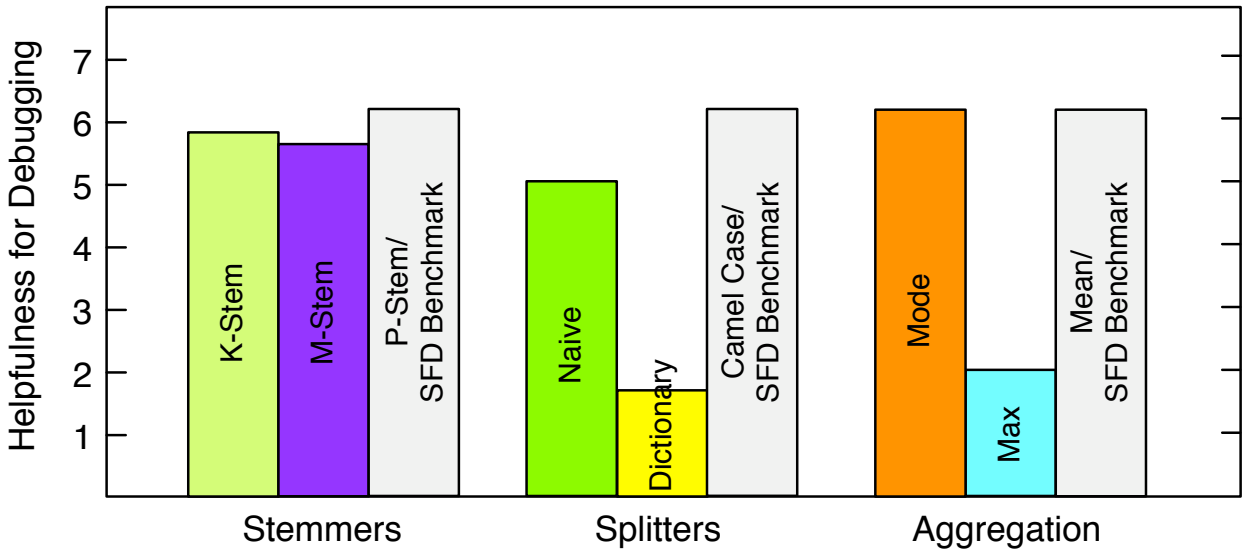

Figure 10.3: Experiment 2's user-study results assessing diagnosis helpfulness for debugging. Higher scores indicate more helpful diagnoses.

In addition to the quantitative Likert-scale surveyed responses, I also gathered qualitative think-aloud impressions of the subjects. When trends were observed in the qualitative results, I highlight the findings.

For the stemmer techniques, the mean for P-Stem is slightly higher than the rest, however the difference among all stemmers is statistically insignificant. Qualitatively, a common concern that was expressed regarding the stemmers' effects on the diagnoses was the subjects' ability to determine the root, unstemmed word. For example, one subject expressed, "I don't know what 'kei' is... maybe keyboard...." However, in almost all cases, the developers were able to correctly guess the unstemmed word.

For the splitter techniques, the enhanced camel-case splitter provided the highest quality and most helpful diagnoses. The naive splitter performed slightly above neutral. Utilizing a paired t-test, I found that the enhanced camel-case splitter performed significantly better than the naive splitter, with a p-value of 0.01. Qualitatively, the splitter technique greatly influenced the semantic meaning for the subjects, because the splitter determines which

words MANTIS can return. Of note is the poor performance of the dictionary splitter, which indicates that the subjects perceived the dictionary-based results as being of poor quality and unhelpful for debugging. For example, during the evaluation of the dictionary splitter, the subjects were often confused because the words had nothing to do with the program (e.g., "veal").

For the aggregation statistic, the mean and mode statistics performed quite well, whereas the max performed quite poorly. In a paired t-test between the mode and mean, I found no statistically-significant difference between their scores (the p-value was over 0.2). The max statistic caused the SFD diagnosis to contain terms that were irrelevant to the fault. Qualitatively, the subjects expressed comments about the max-configured MANTIS such as, "This just looks like you took the entire occurrence of the text and threw it at me," and, "its just saying words that are in the program."

Additionally, the "helpfulness" results in Figure 10.3 are uniformly, slightly higher in all categories than the "quality" results in Figure 10.2. Qualitatively, the subjects often commented that while a result was not completely accurate, it, "tells [them] where to start," and informs them of, "what [they] are looking for." Such comments explain how it is possible aid fault comprehension even if the diagnoses are not perfect.

10.3 Analysis

In this section we present a discussion encompassing both sub-experiments to draw more general conclusions. I discuss each research question individually then consider them all together.

$H4_1$: Optimizing text parsing provides higher quality Semantic Fault Diagnoses.

For some configurations, the results of the two experiments are seemingly contradictory. In sub-Experiment 1, we found that the simple splitting — naive — give results that are insignificantly different enhanced camel case. However in the user study, we found that naive performed statistically worse.

I first investigated the naive splitter in the context of AspectJ and found a clear explanation: these words describe AspectJ and the behavior of the system in general (and are expected to be present in a bug report), but often do not describe a particular fault. For example, in AspectJ the results include terms like "aspect," "compiler," or "weaver." These words are likely to be of little value to developers performing fault comprehension. This is reinforced by comments from the expert judges which suggested that the naive splitter just, "indicates the failure but not the behavior," or that, "its just saying words that are in the program," or "[I] dont think it describes the failure behavior very much." Although the dictionary splitter is more sophisticated than naive, it selects words that are in the dictionary but may be entirely irrelevant to the program or fault domains.

Thus the harmony within these two studies is that the choice of how to split words drastically impacts the results by determining the domain relevance of the words chosen. In some cases, these words still fit within the problem domain, (*i.e.,* naive) but are reduced in effectiveness because they are less helpful or fault specific. As such, for research question H_1 I find:

Dictionary-based and naive splitters produced words that are semantically irrelevant to the fault. Enhanced camel-case splitters produced statistically more valuable words that have a higher semantic relevance to the fault. As such, optimized text parsing provides higher quality results.

$H4_2$: Optimizing text normalization provides higher quality Semantic Fault Diagnoses.

In both studies the choice of stemmer had little impact on the results: all were equally beneficial. There is a slight preference to P-stem, but without more testing, any statistical importance is unclear.

These findings concur with those of Hull and Grefenstette's [37], who found that various stemmers give a similar benefit. In fact, I observed in the user study, and further speculate, that users are able to translate overly-stemmed terms back to the problem domain of the program. For example, diagnoses that contained the term "rat" was correctly interpreted by the human subjects as "rate." I expected stemming to have a larger impact in Experiment 1, however the results are generally within a single percentage point (though P-Stem was statistically more precise than K-Stem).

As such, for research question H_2 I find:

Although stemming was beneficial, different types of stemmers had little impact on effectiveness, all stemmers provided somewhat equal quality in results. As such, optimized normalization does not provide higher quality results.

$H4_3$: Optimizing dimensional-reduction techniques provides higher quality Semantic Fault Diagnoses.

During the investigation into the aggregation statistics, similar phenomenon to the splitters was observed. *Max* produces diagnoses containing terms that describe the program, but only loosely describe the fault, if at all — this behavior is demonstrated in Figure 10.4. When using the max statistic, meaningful correlations between passing and failing test cases and source-code execution are lost due to the fact that only a single data point is used. In other words, using a statistic like max fails to take advantage of the benefits of statistical fault-localization's failure correlation and has poor statistical inferencing power. However, both mean and mode statistics use the scores across the whole program, and perform comparably well.

In addition, most of the experts in the user study were unable to distinguish between the two. More than any other pair of techniques the users often found the results to be nearly identical. In fact, some users even asked questions like, "[is this] the same?"

Thus, it appears that while some optimization are important in improving results, these results suggest that there may be a ceiling of quality that can be achieved with further optimizations providing diminishing returns. As such, for research question $H4_3$ I find:

The max statistic produced terms that are semantically irrelevant to the fault. Mean and mode produced terms that are semantically relevant to the fault. Optimizations in dimensional-reduction techniques do lead to higher quality results, though there appears to be a threshold.

$H4_4$: Optimizing the fault localization produces more fault-specific results.

This result had the most dramatic difference in quality of result. While only performed in sub-experiment one, the results demonstrated an order of magnitude difference. Indeed, performing fault localization with a random correlation algorithm mean that precision dropped over 60%.

A deeper investigation found that random was only selecting words from the source, and so only could describe the domain generally, and even then, could not guarantee that any single feature of the domain was being described (as opposed to a smattering of cross cutting features). As such, these results suggest that the quality of the fault localization appears to have the single largest impact upon the quality of the results.

As such, for research question $H4_4$ I find:

Random correlation is unable to provide words which are even domain relevant to the fault. Optimizations in fault localization produces more fault-specific results.

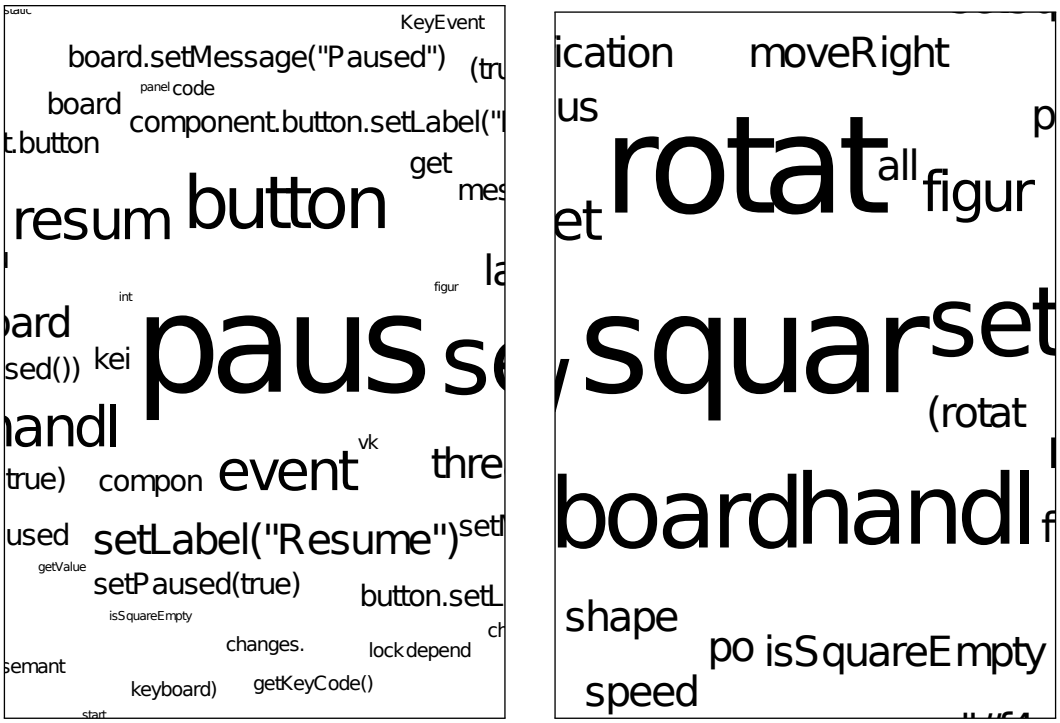

Figure 10.4: Relevant and irrelevant diagnoses (left and right, respectively) for a bug that is activated when the pause button is pressed. Left uses the *mode* aggregation statistic, and right uses *max*.

Overall Interpretation of the Results

The alteration of parsing, correlation or fault localization appears to have a significant impact on the results. The largest difference was that the words provided were either relevant to the program but irrelevant to the fault or relevant to both. This can be seen in Figure 10.4. This figure shows real MANTIS results for one of the bugs in TETRIS. The fault is that the pause button does not stop the figure from moving. The left figure has words like pause, button, event, resume, setLabel("Resume"), isPaused(), and thread.setPaused which clearly point to a problem with the pause feature of the software. The right figure has words like square, new, rotate, board, set, and handle which describe events within the TETRIS program, but are not relevant to the pause functionality. Thus, in these sub-experiments, the more advanced, software-specific/statistical oriented techniques performed more effectively and produced more semantically meaningful words.

When considered as a whole, this experiment demonstrates that while all the steps in the workflow are important, that different methodologies for performing a task may or may not impact the final results in a significant fashion, or only may be able to improve the results to a certain degree. It also outlines that some steps are more critical to the final results than others. For example, changing the fault localization metric had a more significant impact than any of the natural language steps, and within the natural language pipeline, the parse step saw the most varied results. However, the results also showed that there appeared to be no meaningful difference between stemmers in terms of overall impact on result quality. While each methodology to normalize the terms was different (and indeed some performed better for some faults and worse for others) when considered across all the faults, they performed statistically the same. As such, for hypothesis *H4* I find:

> Optimizations for the text parsing, dimensional-reduction techniques, and fault localization resulted in higher quality results. However, optimizations for text normalization provided no statistical difference in result quality. As such, this hypothesis is invalidated.

10.4 Threats to Validity

As the two sub experiments described in this section are expansions upon earlier designs (see the experiments for hypothesis *H1* and *H2*) their threats are largely similar. As such, I only present a brief summary of their weaknesses, and a full treatment will not be recounted here in detail, but the reader is referred to those sections for a more complete treatment.

These experiments have a threat to external validity in that in each experiment, only a single subject program is used (either AspectJ or Tetris). A further concern for external validity

is that only a small group of five experts for the user study, meaning that this group lacks the size to contain meaningful statistical representation. A concern for construct validity is that the TETRIS game uses hand-seeded faults and not "real" faults discovered during development. A final concern with construct validity is that the ASPECTJ study has no meaningful measure of "usefulness" as precision is only a measure of whether the words were also chosen by developers.

While these concerns are real and scope the impact of the results of this study, it is of significance that these two studies somewhat account for the weaknesses of the other. For example, the ASPECTJ study uses only real faults discovered during development (accounting for the hand-seeded faults within TETRIS). A final example is that the lack of observable quality within precision metric is accounted for in the user study where real experts assessed the quality of the results. As such, while these studies have limitations, their design is such that each study somewhat accounts for the weaknesses in the other resulting in a more complete and full understanding of the hypotheses.

Chapter 11

Hypothesis H5

H5: Semantic Fault Diagnoses improve developer performance while debugging.

- $H5_1$: *Do developers leveraging Semantic Fault Diagnoses find the fault more often?*
- $H5_2$: *Do developers leveraging Semantic Fault Diagnoses fix the fault faster or more often?*
- $H5_3$: *Are there differences between debugging complex and simple programs in terms of improvements imparted by Semantic Fault Diagnoses?*

11.1 Experimental Design

The summarization of the experimental design, the hypotheses addressed, and the benefits and limitations are given in Table A.2.

The goal of this experiment is to assess the impact of utilizing MANTIS during actual debug-

Table 11.1: Summary of Experimental Design.

Hypotheses Addresssed	*H5:* Semantic Fault Diagnoses improve developer performance while debugging. $H5_1$: *Do developers leveraging Semantic Fault Diagnoses find the fault more often?* $H5_2$:,*Do developers leveraging Semantic Fault Diagnoses fix the fault faster or more often?* $H5_3$: *Are there differences between debugging complex and simple programs in terms of improvements imparted by Semantic Fault Diagnoses?*	
Object of Analysis	NanoXML (7+ KLOCs, 4 mutants) and Space Invaders (1 KLOC, 4 hand seeded faults)	
Independent Variables	**I1.** Debugging Aid	$i1_a$. Mantis assistance $i1_b$. No automated assistance
Dependent Variable	**I2.** Time	$i2_a$. Time to find bug $i2_b$. Time to complete task (*i.e.,* fix bug)
	I3. Mantis use	$i3_a$. Use of concordance before debugging $i3_b$. Use of concordance during debugging $i3_c$. Use of Mantis before debugging $i4_a$. Number of times Mantis was used $i4_b$. Time duration for each Mantis use
	I4. Task Completion	$i4_c$. Did participants find and fix the fault
Benefits	**B1.** Observe users interact with and leverage SFD **B2.** Assessment of SFD during actual debugging **B3.** Evaluation of practical impact of SFD upon debugging	
Limitations	**L1.** Due to limited number of subjects and bugs, reduced generalizability **L2.** Because every user is different, reduced repeatability	

ging. As such, the experiment is designed to enable not only quantitative answers like, are users statistically more likely to fix a bug when given Mantis, but also a more qualitative understanding like, how often did users leverage Mantis, and in what ways did it change the way they normally would have debugged. The variables leveraged in this experiment reflect this design.

Independent Variable 1: Debugging Aid.

In order to determine SFD improves developers performance during debugging, I leverage the standard "a-b" testing approach where developers have one of two aids:

Mantis assistance: Developers have a Mantis to reference and use during debugging in addition to the Eclipse IDE.

No automated assistance: Developers have only the Eclipse IDE, they do not have access to Mantis.

Dependent Variable 1: Time.

One important signal to determine improvement is the time taken to achieve various "milestones" during debugging.

Time to find the bug: Time to find the bug, and know they have found it.

Time to complete task (*i.e.*, fix bug): Time taken to fix the bug.

Dependent Variable 2: Mantis use.

Use of concordance before debugging: Did the developer use the concordance before starting to debug.

Use of concordance debugging debugging: Did the developer use the concordance during debugging.

Use of Mantis before debugging: Did the developer use MANTIS before starting to debug.

Number of times Mantis was used: The number of time a user used MANTIS in any way during debugging.

Time duration for each Mantis use: How much time was spent during each use of MANTIS.

Dependent Variable 3: Task Completion.

Task Completion: Was the user able to find and fix the bug.

Dependent Variable 4: User Survey.

Found Mantis Harmful: Did the user find Mantis harmful overall.

Found Mantis Useful Before Debugging: Did the user find Mantis harmful if used prior to looking at the code.

Found Mantis Useful During Debugging: Did the user find Mantis useful during debugging.

Task Difficulty: How difficult did the user find this task.

Years of Experience Programming: How many years of experience does the user have with programming.

Used Concordance: Did the user utilize the concordance feature?

Found Concordance Harmful: Did the user find the concordance feature harmful overall.

Found Transparency Feature Helpful: Did the user find the transparency feature helpful.

Found Relational-Words Feature Helpful: Did the user find the relational-words feature helpful.

The high level outline of the user study is that first, all participants individually gain training with MANTIS (including its concordance feature), second, the users are shown the game Space Invaders, third, the users are explained that they will perform two tasks, what they are supposed to do, the time constraints, their goal and the reward system, fourth they would perform the tasks, and finally they would be surveyed and interviewed. I discuss each step in sequence to more clearly illustrate how it took place.

First, all participants were individually given training on MANTIS with its respective features. This step is vital to the success of the study as an inadequate understanding of how to use the tool directly biases the validity of the results. As such, the researchers walks through MANTIS, what it is, what the words mean and the explaining the relationships of size and location within the Buggle. The researchers then walked through a practice debugging example with a program that is not either NanoXML or SpaceInvaders, in this case Tetris was used as it is simple enough so the participants can easily follow. The goal of the researcher demonstrating this is to enable the participant to understand how MANTIS can help them debug. As such, the researcher showed the results, interrupted them out loud, went to the code and used MANTIS as a guide to find the bug, and fix it. During this time the researcher leverageed the concordance feature to ensure its usage is clear. Once this example has been shown, the researcher tested the participant by asking them various questions about MANTIS usage such as: "Why is MANTIS showing you these words in this example; How do I find which words relate to the word $< X >$; and Once I have click on $< X >$ how do I reset my picture so as to stop using the concordance." The goal of these questions was to ascertain that the participant has truly understood how to use MANTIS and its respective features,

and if there are questions, extra time should be taken to clarify its usage. This training took between 15 and 20 minutes.

The second step was to show the users the game Space Invaders. The goal here was to ensure that they are familiar with this game, the goals, how it works, and its functionality. As this program is going to be used during the tasks to represent a simpler program the participant is familiar with, it is important that they have no questions as to how it works, what a user might do, and the respective use cases for this game. If they are totally unfamiliar, allow them to play it for a few minutes to familiarize themselves with it. As this game is very straight forward and it has been chosen specifically because many people are already familiar with it, this training always took less than five minutes.

The third step is to explain to the users the format of the study. The tasks for this study were broken into two stages, which are given to participants in a random order to reduce ordering bias on the results. Participants debugged two programs, each containing a single bug. The programs are NanoXML, an XML compiler with text based input and text based output, and Space Invaders, a GUI based game requiring real time user input. For both programs there is a test suite of input and expected output given to the participants. For one of these programs the participant was able to leverage Mantis, and for the other they were not. Which program they use Mantis with is determined by the researcher such that in the end, there is an equal number of participants who used Mantis and did not use Mantis for each program. Further, the participants are requested to "speak aloud" while they are debugging to explain what they doing, looking for, and trying to understand. To help motivate the participants to be as efficient as possible, they were informed that there was a cash bonus for completing each task more quickly, such that the faster they complete the task, the more bonus money they receive. For each task a user is given a maximum of 25 minutes to find and fix the fault, at which point the task will end regardless of where they are. After each task they were asked to fill out a brief questionnaire assessing how difficult

they thought the problem was. If they used MANTIS during that task, it will additionally ask them whether they thought it was useful or harmful, and to what extent. After they finish both tasks, there will be a brief time where the researcher will ask any final clarification questions about their behavior or experience.

The fourth step was to allow the participant to perform the tasks and to take notes in accordance with the variables described above. This stage was fairly straight forward, but it was key that the researcher take clear and complete notes about the behavior of the participant.

The fifth and final step was to have the user fill out a survey regarding their opinion on MANTIS and ask a few exit questions for final feedback. As mentioned, the debriefing is to allow any final clarifications. The participant is then paid for their assistance, and thanked. Some example debrief questions include:

- What were you looking for in the MANTIS words?
- Did you feel like you found what you were looking for?
- Was any aspect if MANTIS unhelpful towards debugging?
- How did the concordance impact your understanding?
- How did you use the large-font-sized MANTIS words?
- How did you use the small-font-sized MANTIS words?
- Was what you were looking for in MANTIS different during debugging than it was before you started debugging?
- What triggered you to stop looking at MANTIS and go back to debugging?

As a side note, each participant's full study with training and the final debrief took roughly 1.75 – 2 hours.

For this study, I selected 32 individuals from UC Irvine. The participants were required to be a computer science major or a grad student in a computer science disciple (or some variant thereof, *i.e.,* software engineering, or informatics). Further, a participant must have at least two years of experience coding in Java, and have at least one year of experience using the Eclipse IDE (all participants will have to go through a pre-study screening). There was a solid mix of undergrad, grad, and individuals with professional experience providing a well rounded set of candidates reducing any bias coming from the participant sample. Also, the researcher ensured a good mix of students who have had training at other universities or in other countries to create a more generalizable participant pool.

11.2 Results

Note that the tables containing all the values for each participant across the user studies are given in Appendix A. In this section I present the summary and results of statistical testing upon that data, but it is provided for completeness sake. I will present the results as grouped by each dependent variable.

The first dependent variable considered in this user study was that of time, primarily, time to find and time to fix the fault. In this case, note that any participant failing to complete the task was given the time of 25 minutes (the max time to complete the task). This data is shown in Table 11.2 with the results separated for NanoXML and Space Invaders, and then considered with both subjects combined.

This table illustrates some interesting results, first that Mantis does significantly improve the participants changes to find a bug in general, and in specifically the harder tasks (*i.e.,*

Table 11.2: P values for t-test comparing participants who succeeded and those who failed across variable time. (*i.e.,* a p value less than 0.05 means MANTIS statistically significantly improved their performance.)

	NanoXML	Space Invaders	Combined
Time to Find Bug	$p < 0.01$	$p > 0.10$	$p < 0.05$
Time to complete Task	$p > 0.10$	$p > 0.10$	$p = 0.10$

NanoXML) but not specifically in the easy tasks (*i.e.,* space invaders). Additionally, the data shows that for either subject individually, MANTIS does not statistically significantly improve their time to complete the tasks, but when considered as a whole, it nearly does. Indeed, in small sample sizes statistical significance is sometimes alloted to p-values of 0.10. In the case of this study with 32 participants (meaning only 16 participants with and without MANTIS per subject program), that standard implies statistical significance for this value.

The results for the second dependent variable — MANTIS use — are given in Table 11.3. This variable analyzes whether participants used MANTIS before and during debugging, whether they used the concordance feature before and during debugging, and the amount of time they used MANTIS each time it was used. In this case, the only statistically significant value is found in the tinal facet of this variable, time duration of MANTIS use. What this means, is that there was no statistical difference between users who succeeded or failed the task in terms of when they used the concordance feature or when they used MANTIS. However, the data shows that when considering only hard tasks (*i.e.,* NanoXML) or when considering all tasks, the way in which MANTIS is used, specifically the time spent using it at each interval, did have a statistically significant impact. This is in contrast to the numbers when considering only easy tasks (*i.e.,* Space Invaders) wherein there was no statistically significant impact.

To more clearly understand the difference between usage, I present Table 11.4. For this table the information is broken down between each subject, and pass and fail. Since the previous table informs us that these values are statistically significantly different, the goal of this table

Table 11.3: P values for t-test comparing participants who succeeded and those who failed across the variable MANTIS use. (*i.e.,* a p value less than 0.05 means MANTIS statistically significantly improved their performance.)

	NanoXML	Space Invaders	Combined
Use of Concordance before debugging	p > 0.10	p > 0.10	p > 0.10
Use of Concordance during debugging	p > 0.10	p > 0.10	p > 0.10
Use of MANTIS before debugging	p > 0.10	p > 0.10	p > 0.10
Use of MANTIS during debugging	p > 0.10	p > 0.10	p > 0.10
Time duration of MANTIS use	p < 0.01	p > 0.10	p < 0.01

Table 11.4: More detailed data about time duration of MANTIS use.)

	NanoXML		Space Invaders		Combined	
	Success	Failed	Success	failed	Success	Failed
Number of MANTIS uses	31	103	45	11	76	114
Total Time Spent	176	1632	384	115	560	1747
Average Time Spent	5.68	15.85	8.53	10.46	7.37	15.32

is to provide a clearer picture in the usage patterns that represent this difference. In this case, we see that for the harder task (*i.e.,* NanonXML) and when considered in total, that successful MANTIS users were those who used it for a shorter period of time, and typically less frequently. More specifically, if a user spent more time consulting MANTIS, either in terms of frequency of usage, or in terms of time spent each use, they became statistically less likely to complete the task.

The third dependent variable considered was task compeltion, the results of which are found in Table 11.5. In this case, the results show that MANTIS makes no statistically significant difference in the ability to complete the task regardless of task difficulty.

The final variable considered is that of user opinion as determined by a survey. In this

Table 11.5: P values for t-test comparing participants who succeeded and failed across the variable task completion. (*i.e.,* a p value less than 0.05 means MANTIS statistically significantly improved their performance.)

	NanoXML	Space Invaders	Combined
Task completion	p > 0.10	p > 0.10	p > 0.10

case, each user was given a survey and a Likert scale ranging from -3 to 3 (with -3 being completely disagree and 3 being completely agree). The results are broken up by subject and are presented in Figure 11.1 for Space Invaders and Figure 11.2 for NanoXML.

In terms of task difficulty, uniformly the users found NanoXML harder (regardless of fault). Further, the data shows that when given MANTIS, the users found the task at least as easy or easier than if they didn't. Overall the users also found MANTIS to not be harmful to their debugging, though those who failed the task found it more slighly more harmful than those who succeeded. Further, whether participants succeeded or failed they all found MANTIS to be useful to them prior to starting to debug. This changes somewhat during debugging where all participants still found it useful, but those who failed the tasks found it less so.

When considering specific MANTIS features (*i.e.,* concordance, transparency and related words) the participants found the concordance to not be harmful, but again those who failed the tasks found it slighly moreso than those who succeeded. However there was some disagreement between subjects regarding the transparency feature, when considering just Space Invaders, the users found it harmful, whereas with NanoXML, they did not (though those who failed the task in NanoXML found it neutral). Lastly when considering the related words, all the participants found this feature harmful save those who succeeded on Space Invaders, who found it neutral.

Lastly, it is of note that there was no statistically significant difference in terms of years of programming experience between those who succeeded or failed tasks in either NanoXML or Space Invaders.

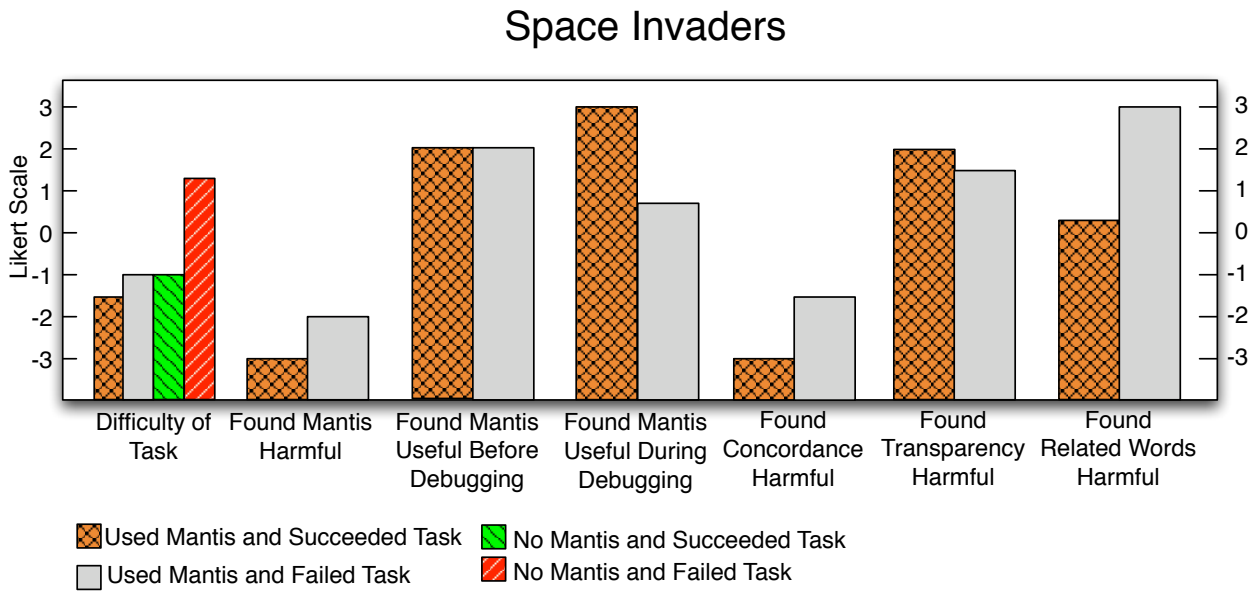

Figure 11.1: The Mode value given by participants in their assessment of MANTIS after completing a Space Invaders task.

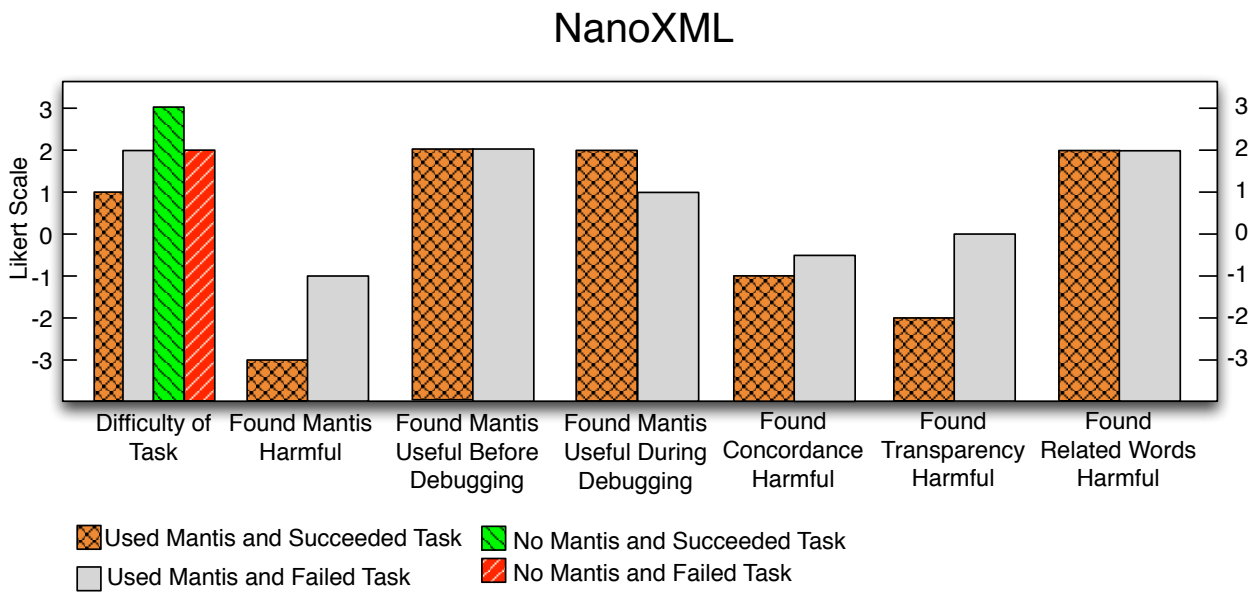

Figure 11.2: The Mode value given by participants in their assessment of MANTIS after completing a NanoXML task.

11.3 Analysis

In this section I present the corresponding discussion to the results previously presented, in that the results will be discussed in the context of the research questions presented previously, and also take into consideration some of the qualitative responses from users.

$H5_1$: Do developers leveraging Semantic Fault Diagnoses will find the fault more often?

To be classified as "finding the fault" the participant had to both actually arrive at the fault location in the code, and then signify that they were at the fault; in other words they had to find it and then know that they had found it. This means that the participants had to have some understanding of what the fault is, but not necessarily enough to fix it. The results show that overall, Mantis does improve a developers ability to find the fault, both in terms of whether the participant was able to find it at all and in terms of how fast they were able to find it. However, when only the simple tasks were considered (*i.e.,* Space Invaders) there was no statistically significant improvement.

When considering their think aloud statements, most participants who had Mantis seemed to use it as a type of guidepost, or sounding board to reinforce their confidence that they were on the right track or help them know what in the code they were looking for. Participants would see the keywords in Mantis, and then formulate an idea about what they were looking for, and then reference back to Mantis occasionally to make sure as one participant put it, they were, "on the right track."

As such for research question $H5_1$, I assess:

Using the semantic fault diagnosis, the subjects were statistically significantly more likely to find the fault, and do so in a faster time.

$H5_2$: Do developers leveraging Semantic Fault Diagnoses fix the fault faster or more often?

The numbers presented in Table 11.3 demonstrate that Mantis does not statistically improve the ability of developers to fix a fault faster when strictly considering a 0.5 p value for statistical significance. However, in many cases where the sample size is small (as is the case for this study where for each program there were only 16 participants on each side) a p value of 0.10 is often used to denote suggested significance. The idea being that if the sample size were larger, it would likely be significant. If this measurement were used, indeed we would see that Mantis does statistically improve a developers chances to fix a fault when consering the programs together, and improve their speed in doing so.

It should also be noted that out of all 16 participants who had NanoXML tasks but not Mantis, only one was able to fix the fault. This is in contrast to the four who were able to fix it with Mantis (and increase of 19%). However, when considering only Space Invaders, there is the same number of successes and failures for participants who had Mantis and those who didn't.

When considering their think aloud comments in addition to the numbers from above along with their usage patterns, some interesting characteristics were observed. First, is that the participants who succeeded seemed to trust Mantis, meaning that they continued to stick around the functionality it described, even if they weren't immediately successful in finding a fault. This was critical for NanoXML as the code was quite complex and it wasn't immediately obvious what was wrong. Participants tended to need time to go through that code, and those participants who stuck with it, were more successful. Some even remarked

that while they didn't see a fault, it must be there because MANTIS was describing it.

This is in contrast to participants who didn't trust it as much. These users often seemed to spend more time with MANTIS, tryign to analyze each individual little word and scrutinize whether it should be there. These users often tried to simply try and find some word, any word that reinforced what they believed, regardless of how small or how few times it appeared in MANTIS's description.

Another facet of this question is whether MANTIS makes it easier to fix these faults. As such, understanding how difficult the users felt each task was is important. Remember that given MANTIS, participants felt the tasks were easier, even if they failed them. One interpretaion of this is that MANTIS provided a more complete context so the users knew what they were looking for and why, even if they did not have the technical skill to fix it. This implies a limitation of MANTIS that even if it can guide a developer to the functionality that is broken and help them know why its broken, it cannot impart sufficient understanding to know how exactly to fix it.

It is also interesting of note that participants found that while the concordance feature was not harmful, the related words and trasnparency feature were. The participants suggested that tfor the transparency, it obfuscated too much of the relationships away, and as one user put said, "it nice in theory but it removed connections I needed." There was a consensus amoung participants that too many of the connections were hidden when using this feature which led to an incomplete picture of what was going on. However, most participants tried this feature, which implies that this is something they want (the ability to summarize a specific connection) but that MANTIS's implementation was too aggresive. As for the related words features, the users almost universally said that they wanted to know the locations of those words in the code. For example, a user clicking on the "reader" word would get a list of various terms related to this, but have no context for them. Users suggested that the words come with a location (*e.g.,* class, line number, or method name) to help them have

a little more context.

As such for research question $H5_2$, I assess:

> Using the semantic fault diagnosis, the subjects were statistically significantly more likely to fix the fault, and do so in a faster time for harder tasks.

$H5_3$: Are there differences between debugging complex and simple programs in terms of improvements imparted by Semantic Fault Diagnoses?

The goal in understanding this question is to know when Mantis can be expected to imparts the benefits seen in the previous questions. In this case, the study had two programs, one that was hard to fix — NanoXML — and one that was easy to fix — Space Invaders. The data shows that fairly uniformly regardless of the question asked, Mantis performed much better with harder more complex tasks, and imparted smaller if any benefits on easier tasks. This can be seen across a few areas, first that the same number of participants fixed the Space Invaders faults whether they had Mantis or not, and that there was no statistically significant difference when considering just those who failed in whether they could find the fault between Mantis users and non-Mantis users. Further, the time to fix faults was no improved when considering strictly the Space Invaders faults. However, when considering these exact same measurements for NanoXML, we find that Mantis does make a difference in these cases. These results suggest that Mantis is not as effective when considering easy tasks.

This is also reinforced by the qualitative assessments from the participants. Many of the participants said that they thought the Space Invaders faults were so easy that they didn't really need assistance (for reference the mean fix time was seven minutes with Mantis and

nine minutes without). Most users were able to figure out the fault without any assisance by simply running the game and seeing the fault in action. This is in contrast to NanoXML where the output was a compiled XML text and the correlation bewteen input and output was much more complex. Users typically used MANTIS less with the simplier program, even when they had it because one user said, "I didn't need it." In these cases, the context was easily identifiable by observing the program, and thus MANTIS was unable to impart additional context to hep them.

As such for research question $H5_3$, I assess:

> Semantic fault diagnosis is mostly helpful for harder more complex problems, providing little to no assistance on more simple tasks.

Overall Interpretation of Results

As a whole this user study provides a great deal of insight into the impact MANTIS can have during debugging. At a high level, it appears that MANTIS significantly improves the chances that a developer can find a bug, and that they can find it more quickly, and suggests that it can also improve their ability to fix it. However, the data demonstrates that this is only true for more complex problems where the context of the fault cannot be immediately inferred from the test case, or by running hte program.

At a more detailed level, the value of MANTIS seems to have a strong correlation with its use as developers with similar backgrounds who self-reported difficulty of tasks equally varied in their success based upon how they used MANTIS. Those participants who used MANTIS as a type of roadmap, guiding them to functionality and as a reference were significantly more successful than those who spent a great deal of time trying to analyze each word MANTIS provided. Further, the results clearly identify that MANTIS has room to improve in its

ability to clearly identify the nature of the fix. Participants reported that they were able to understand the context of the fault, and in many cases know what they were looking for that was broken, but were mostly left unaided in their task to make changes to fix said fault. In this way, MANTIS needs to improve in its ability to provide information from developer comments which identify the task of what the code should be doing, so developers more clearly can see the difference between expected functionality and desired functionality.

It is also of note that even when tasks were failed the participants rated the tasks as more easy when they had access to MANTIS. This suggests that MANTIS is able to reduce some cognitive burden and provide some guidance into what the problem is, even if the developer cannot fix the fault.

As such for hypothesis $H5$, I assess:

> MANTIS was able to improve a developer's ability to find and to some extent fix faults for harder tasks, though MANTIS was unable to improve performance for simple tasks. Further, developers found problems easier when given access to MANTIS, even if they were unable to fix them. As such, the hypothesis is not invalidated.

11.4 Threats to Validity

While this study will provide a variety of valuable information to the research agenda, it is not without its shortcomings. One non-trivial threat lies with its external validity. All user studies are somewhat biased based upon the participants used, the number of them, their background, and a number of other factors. As such, this study cannot completely generalize as its participants will all be found at UC Irvine and will thus share to some extent a similar background. However, notwithstanding the similarity within some participants, this

study will leverage over thirty candidates, which is a significant amount when compared to existing work within the software engineering research community. Further, care will be taken to ensure participants come from a variety of background, including different countries, different levels of experience and different levels of industrial experience to attempt to reduce this concern.

Another threat to the external validity of this work is that only two subject programs are used within the study. As such, the researcher cannot make concrete claims about whether the results of the study would apply to the large variety of software within the world today. In an effort to reduce this concern, the design of this experiment uses two types of programs, one with a GUI interface with few features, and another that is strictly text based with many different features. While this does not completely counteract this concern, it does suggest that if the results hold across these two programs, they may well hold for others.

Chapter 12

Summary

Now that the results from investigating the hypothesis have been discussed individually, I briefly discuss their impact when considered as a whole. To summarize the previous findings, we were unable to invalidate hypothesis *H1, H2, H3* and *H5* but we did invalidate *H4*. Remember that in the case of *H4* only a single research question found negative results, and thus it would be possible to reconstruct *H4* such that it ignored term normalization, which would result in an invalidated *H4*.

As such, the conclusions drawn from the non-invalidated hypotheses are that: (1) SFD describes faults with similar keywords to developers, (2) SFD produces quality (*i.e.,* fault specific and helpful) fault descriptions, (3) SFD reduces cognitive burden on developers more than existing locational based techniques, and (4) SFD improves developer performance when considering difficult tasks. As the ultimate goal of this research agenda is to answer the question of whether SFD will assist developer comprehension during debugging (*i.e.,* Hypothesis *Hyp*), we will consider these results in the context of that space.

There is evidence both in industrial practice and academic research by Bettenburg *et al.* suggests that a quality bug report helps developers understand and increases the likelihood

of the fault being able to be fixed [9]. While at the beginning of the debugging phase, specifically during the contextual understanding portion, this role is significant in the debugging process. Thus, knowing whether SFD describes faults using similar key words as developers approximates asking whether SFD describes a bug similarly to a bug report, albeit in a summarized way. As the results demonstrated earlier, for my experiments this was found to be the case, suggesting that SFD results may also help developers understand and increase the likelihood of them fixing their fault. Further, the second experiment found that real developers thought the descriptions given by SFD were both fault specific (*i.e.,* they specified accurately the fault behavior) and that they would be helpful for developers. These then are two pieces of evidence suggesting that SFD assists developer comprehension during debugging, particularly during the early phase.

While these two pieces of evidence are significant, if existing tools were better at assisting developers, or more efficient in reducing their cognitive burden, then there would be no need for SFD. However, during a direct comparative study, the earlier results found that SFD took less time to interpret, enabled a clearer understanding of the fault, and was unanimously voted more helpful than existing locational based tools. While only a single study, this result is encouraging that SFD may assist developers better at understanding a fault than existing techniques.

The goal of these hypothesis was in part to gauge whether SFD was able to accurately describe faults, and whether a user interpreting those results found them to be helpful. All the studies have approximated the ability of SFD to benefit developers and have sought to understand in what ways SFD is helpful and at what point in the process of debugging SFD can provide value. When considered together, these three data points suggest that SFD can aid in developer comprehension of the fault, can be helpful, and is more effective at rendering help for this task than existing fault localization techniques; particularly at the beginning of the debugging process.

The previous data points then explain much of what was seen durig the final experiment where a developer had to sit down and attempt to find and fix code. Developers uniformly stated they thought MANTIS was helpful, particularly prior to making edits in the code as an aid to form an initial impression and get an idea of the functionality involved with the fault. Further, I found that when used as a type of roadmap, MANTIS was able to improve a developers ability to find the fault, know they had found it, and then fix it. In addition to this type of quantitative assistance, my results found that given SFD, developers thought that their tasks were easier. This piece of evidence in cruicial in considering whether SFD is able to reduce cognitive burden as it is a subjective problem, as this data implies that using SFD makes developers feel the problem isn't as hard, even if they are unable to fix it. While cognitive burden may be impossible to quantitatively measure, the personal feeling of difficulty of task being reduced is a substantial feature of SFD that implies it is being reduced.

It is significant however that the ability of SFD to provide assistance was largely affected by the difficulty of the problem presented: for simple tasks wherein the programs context was very clear, SFD was able to provide little to no assistance. The user study found that in those cases users simply did not need SFD as it was unable to provide additional context to what they had already gleaned from observing the program in action.

When considered as a whole, the initial studies reinforce and help explain the final findings in the user study, and the user study demonstrates in practice that the approimations found in the easlier studies were valid. While the debugging task contains many stages from initial conceptualization to making changes in the code base, these studies have examined how SFD can fit into this task to impact performance and have uniformly agreed that its impact is positive.

Indeed, when considered together these results suggest that SFD can aid developers in comprehension of their fault, can be helpful, is more useful than existing work, can be optimized

with future improvements, and will enable enhanced performance during actual debugging.

Chapter 13

Related Work

To my knowledge, the concept of performing automated semantic fault diagnosis is new. However, similarities in technique and concept can be found in many existing research endeavors. In this section, I discuss representatives of these works in five categories: program comprehension, statistical fault localization, fault contextualization, software reconnaissance and code summarization.

13.1 Program Comprehension

A number of research endeavors involve source code string matching for the purpose of enabling querying. Revelle and Poshyvanyk [67], Hill *et al.* [35], and Shepard *et al.* [71] provide facilities and studies that provide useful and meaningful textual results based on queries. Mantis shares similar techniques to these past approaches in that both match, index, and extract strings from the source code. However, two substantial differences are that my work requires no user input or expertise to invoke, and in addition, incorporates developer comments to describe run time events. More precisely, I expect Mantis to be invoked

when the users initially encounter the failure, meaning that they do not have sufficient expertise to create a query, and there are no bug reports from which to automatically draw queries (because there is no one with expertise to write said report). Further, while existing techniques leverage comments, I utilize them in concert with dynamic information in a way that is unique compared to past techniques.

Another body of research involves feature or concept extraction from source code. One survey of this work was done by Dit *et al.* [25] that examines natural language, dynamic and web-based information retrieval techniques. Specific examples of such approaches include Ohba and Gondow [57], Maletic and Marcus [51, 52], Kuhn *et al.* [43], Grant *et al.* [31], Zhao *et al.* [86], and others (e.g., [26, 34, 59, 64, 65, 76]).

The most common concept extraction approach is to apply techniques such as latent semantic indexing to reduce source code to topics at modular levels (*e.g.,* at the method level) in order to provide descriptions of functionality for segments of code. Similar in concept, in that they utilize words and strings found in the source code and offer abstractions that describe meaning, however, such approaches target a different purpose. As a further point of divergence, many techniques do not consider dynamic information and those that do most often use it as a simple filter upon their final results (*e.g.,* [29, 79]) while my technique leverages dynamic information to describe concepts or meaning for disjoint (*i.e.,* non-modular) code.

13.2 Source Code Normalization

To supplement latent-semantic analysis techniques at the source code level, research has investigated different normalization properties including splitters and stemming. Splitting approaches take an identifier (*e.g.,* `getFoodNow`) and split it based on its constituent in-

dividual natural-language words (*e.g.,* `get, food, now`). Lawrie *et al.* created GenTest, which splits camel cased words, creates a usage dictionary, and uses a traditional English dictionary to expand abbreviations [44]. GenTest allows the transformation of identifiers like "`intScore`" into "`integer, score`" for enhanced readability and searchability. Guerrouj *et al.* enhanced usage dictionaries by creating a splitter called TIDER which splits identifiers by determining their Levenstein distance from words in a traditional English dictionary [32, 50]. These two splitters expand upon domain knowledge usage by using traditional English dictionaries to determine their splitting criteria.

Stemming approaches take an identifier (*e.g.,* `rating`) and reduce the influence of plurality or verb tense by removing/modifying the word's ending (*e.g.,* `rate`). The Porter stemmer [61, 63], Paice stemmer [58] and Lovins stemmer [48] are examples of such techniques. Recently, many researchers have been focusing on morphological stemmers — each word's ending is removed, with a comparison against a dictionary to locate the "true root." Krovetz was one of the earliest to create this type of stemmer [42]. Later his work was expanded upon by Schone and Jurafsky who employed a usage dictionary to find the root which holds the most similar semantic meaning [68]. Even later, Wiese *et al.* created a stemmer called "M-stem," which utilizes a set of manually predefined words with their corresponding stems [81].

A recent study by Binkley *et al.* [11] investigated the impact of these types of normalization techniques upon IR-based approaches within source code and found that they indeed improve precision and recall. Our approach leverages these types of normalization techniques as part of step five in our overall approach (See Section 4).

13.3 Statistical Fault Localization

Research in the area of statistical fault localization is related in that: (1) our technique builds on such techniques, and (2) they also describe aspects of faults. Some examples of such techniques include work by Jones *et al.* [41], Liblit *et al.* [46], Liu *et al.* [47], Wong *et al.* [83], Abreu *et al.* [1], and DiGiuseppe *et al.* [20, 21]. The insight of such techniques is that events that occur frequently during the execution of failing test cases, but rarely during passing test cases, are likely to be the fault causing the failures. In contrast to semantic fault diagnosis, such techniques describe *locations* in the code for a developer to inspect, instead of extracted words for the purpose of describing *meaning*.

Work more similar in approach is research by Lukins *et al.* [49] and Andrzejewski *et al.* [5]. Lukins indexes the source code text, including comments, and then automatically attempts to locate faults based on text found in bug reports. Andrzejewski provides improvements to traditional topic modeling algorithms to partition "usage topics" from "failure topics" in order to locate source-code predicates that predict failure. In each of these techniques, feature extraction techniques are utilized to identify locations in the source code, as opposed to descriptions of the faults.

13.4 Fault Contextualization

A number of researchers have recognized the difficulty in understanding faults based on solely location, as is provided by many statistical fault localization techniques. Jiang and Su [38], Hsu *et al.* [36], Chilimbi *et al.* [12], Babenko *et al.* [8], and Masri *et al.* [54] provide techniques that give locational fault diagnoses in terms of *sequences* or paths of execution, instead of solitary location localization. Similarly, Baah *et al.* [6] and Deng *et al.* [17, 18] provide dependence models of the program that can be used to identify flow paths

that correlate with failure. These techniques attempt to present a fault through program structure (*i.e.,* locations in the code and their relationships). My approach, instead, provides simple natural language, semantic, and topical descriptions.

In summary, I do not foresee nor advocate semantic fault diagnosis replacing any techniques or purposes described here. Instead, I imagine that it can provide developers with descriptions to understand the nature of the fault. With this understanding, developers may be better prepared to query the source code to understand the results of fault localization or contextualization techniques. Moreover, future semantic fault diagnosis techniques are likely to evolve to utilize more advanced localization, contextualization, and program comprehension approaches.

13.5 Software Reconnaissance

Researchers have attempted to correlate run time events with a variety of user defined "features" in the form of a technique called software reconnaissance as described by Wilde *et al.* [62, 82] and Scully *et al.* [69]. Software reconnaissance builds mappings between software features and software components by determining for each test case, whether it executes a specific feature. In this case, the user defines a type of functionality or feature (*e.g.,* call forwarding for a phone system), and then labels each test case as either invoking said functionality or not. Then, each test case is instrumented, and dynamic correlation techniques are applied to between the sets of executed test cases (*i.e.,* those that invoke the functionality and those that do not) to determine which locations of the code are most *relevant* to the feature. In this way, the code most responsible for, or contributing to a feature can be *localized* and identified for users.

This work is similar to fault localization in the circumstance where the user defined scenario

is software failure, and the feature being localized is the fault responsible for failure.

This work is similar to semantic fault diagnosis in that it is very similar to a component leveraged by semantic fault diagnosis (*i.e.,* fault localization). However, software reconnaissance is dissimilar in that it attempts to localize user defined features or components, which requires manual user labeling of each test case, and results in a list of contributing locations. In the case of software reconnaissance, the user already understands the feature to the extent of knowing whether or not its invoked for each test case whereas semantic fault diagnosis attempts to describe a feature (*i.e.,* the failure) to a developer who likely has less mastery of the system. Indeed, semantic fault diagnosis further differs in that it requires no user labeling or input and produces a *description* of a phenomenon (*i.e.,* failure) as opposed to a list of locations.

13.6 Code Summarization and Labeling

Due to the complexities of understanding large bodies of code, researchers have created techniques to summarize, or label sections of code to elucidate comprehension. Sridhara *et al.* created a technique that determines the most *meaningful* code within a method, and then uses that code to a create a summary for each method [73, 74]. This work creates a summary of contiguous section for a specific discrete unit of code (*e.g.,* a method or class) but leverages no dynamic information, as its goal is not to describe run-time events but to identify the most meaningful terms within a method that can be used to describe it. Other work by Lucia *et al.* [15] and Haiduc *et al.* [33] have tried to label discrete contiguous sections of code (*e.g.,* methods or classes) by using natural language processing techniques to identify the top X words within said section of code to best describe it.

While this summarization and labeling work is similar in its use of natural language process-

ing techniques to identify meaningful terms within a larger document, it differs from semantic fault diagnosis in primarily three ways: (1) SFD works by identifying non-contiguous sections of code through dynamic information, (2) SFD describes a run time event as opposed to static code, and (3) the set of use cases for each technique are exclusively dissimilar. Thus, summarization and labeling are similar in their attempt to enable greater comprehension and use of natural language techniques, but dissimilar in their approach, goal, and use of dynamic techniques.

Chapter 14

Contributions

The main contribution of this research agenda outlined in this document is the first set of experiments that would provide direct evidence of the value of semantic fault diagnosis as a debugging aid. This is mainly investigated in five hypotheses, which often contain subhypotheses to require a more fine grained exploration. While these have been discussed in detail individually, and for the work done thus far, collectively, I briefly describe in Table 14.1 the outcome that was gained.

By combining both quantitative and qualitative analysis I was be able to analyze many different facets of how SFD impacts developer comprehension of faults during the entire debugging process. As can be seen, some hypotheses focus on the early stage of debugging, (*e.g.,* attempting to form a first impression, or basic understanding of the fault), whereas others focus on efficiency (*e.g.,* time taken to interpret, or debug) or impact upon capacity to change actual code to fix the fault (*e.g.,* the later stages of debugging). In this way, the outlined experiments present a more complete picture of where, and how SFD may best fit into the debugging process to provide the most value to developers. The experiments analyzing this agenda's hypotheses found that SFD does not exclusively benefit and improve

developer performance, but its ability to assist will be somewhat affected by a variety of factors, including developer usage, underlying complexity of a fault, and other unknowns. As such, this agenda explored these topics to gain a more full picture of how, when, and in what ways SFD can assist developers; and the ways in which it might hinder them.

An additional benefit of these findings is that SFD and its capacity to assist developer during debugging is a new research topic that, to the best of my knowledge, has yet to be explored in existing literature. Thus, this area requires further research, and the methodology used to obtain meaningful evaluations of such an approach serve as a guidance for future work.

Lastly, the agenda implemented provides results that are highly relevant to the research community interested in either natural language usage in software engineering, or debugging. These experiments contribute to existing discussions about the relevancy of natural language in software engineering aids, and what techniques best enable developers to perform some of their most difficult and time consuming tasks. In addition, as these hypotheses were not invalidated and provide results that show more ability to assistance than existing locational based approaches, it provides a baseline for building tools and investigating methodologies to actively support developers.

Table 14.1: Summary of Experimental Contributions. QN is for quantitative, and QL is for qualitative.

Hypothesis	Analysis Type	Outcome
H0	QN + QL	*SFD assists developer comprehension during the debugging process
H1	QN	SFD keywords are similar to developer chosen keywords in fault descriptions
H2	QN + QL	SFD generates quality descriptions
$H2_1$	QN + QL	SFD generates descriptions that are fault specific
$H2_2$	QN + QL	SFD generates descriptions that are helpful to understand a fault
H3	QN + QL	SFD reduces cognitive burden for fault understanding more than existing locational based techniques
$H3_1$	QN	SFD takes less time to interpret than existing locational based techniques
$H3_2$	QN + QL	SFD enables clearer understanding of a fault than existing locational based techniques
$H3_3$	QN	SFD is termed more helpful for understanding a fault than existing locational based techniques
H4	QN + QL	Only certain optimizations in SFD improves the quality of its end results
$H4_1$	QN + QL	Optimizing SFD's text parsing improves the quality of its end results
$H4_2$	QN + QL	Optimizing SFD's text normalization does not impact the end results
$H4_3$	QN + QL	Optimizing SFD's dimensional-reduction techniques improves the quality of its end results
$H4_4$	QN	Optimizing SFD's fault localization improves the quality of its end results
H5	QN + QL	SFD improves developer performance during debugging*
$H5_1$	QN	SFD enables developers to find the fault more often
$H5_2$	QN	SFD enables the finding of the fault faster
$H5_3$	QN	SFD enables the ability to fix the fault more often
$H5_4$	QN + QL	There is a difference between SFD's effectiveness between complex and simple programs

Bibliography

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[2] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[3] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience*, 23(6):589–616, 1993.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, pages 402–411. IEEE, 2005.

[5] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In S. Matwin and D. Mladenic, editors, *European Conference on Machine Learning*, pages 17–21, Warsaw, Poland, 2007.

[6] G. Baah, A. Podgurski, and M. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. *Software Engineering, IEEE Transactions on*, 36:528–545, 2010.

[7] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 73–84. ACM, 2010.

[8] A. Babenko, L. Mariani, and F. Pastore. Ava: Automated interpretation of dynamically detected anomalies. In *Proceedings of International Symposium on Software Testing and Analysis*, 2009.

[9] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008. ACM.

[10] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 482–498. IEEE Computer Society Press, 1993.

[11] D. Binkley, D. Lawrie, and C. Uehlinger. Vocabulary normalization improves ir-based concept location. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 588–591. IEEE, 2012.

[12] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, pages 34–44, 2009.

[13] T. Copeck and S. Szpakowicz. Vocabulary agreement among model summaries and source documents. In *Proceedings of Document Understanding Conference*, 2004.

[14] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press Ltd., London, UK, 1972.

[15] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Using ir methods for labeling source code artifacts: Is it worthwhile? In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 193–202, 2012.

[16] V. Debroy and W. E. Wong. Insights on fault interference for programs with multiple bugs. In *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*, pages 165–174. IEEE, 2009.

[17] F. Deng, N. DiGiuseppe, and J. A. Jones. Constellation visualization: Augmenting program dependence with dynamic information. In *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*, pages 1–8. IEEE, 2011.

[18] F. Deng and J. A. Jones. Inferred dependence coverage to support fault contextualization. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 512–515. IEEE, 2011.

[19] T. Denmat, M. Ducassé, and O. Ridoux. Data mining and cross-checking of execution traces: a re-interpretation of jones, harrold and stasko test information. Technical report, 2005.

[20] N. DiGiuseppe and J. A. Jones. Fault interaction and its repercussions. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 3–12. IEEE, 2011.

[21] N. DiGiuseppe and J. A. Jones. On the influence of multiple faults on coverage-based fault localization. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 210–220, 2011.

[22] N. DiGiuseppe and J. A. Jones. Semantic fault diagnosis: automatic natural-language fault descriptions. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 23. ACM, 2012.

[23] N. DiGiuseppe and J. A. Jones. Software behavior and failure clustering: An empirical study of fault causality. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 191–200. IEEE, 2012.

[24] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[25] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[26] B. Dit, M. Revelle, and D. Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.

[27] L. H. Etzkorn, C. G. Davis, and L. L. Bowen. The language of comments in computer software: A sublanguage of english. *Journal of Pragmatics*, 33(11):1731–1756, 2001.

[28] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30:964–971, 1987.

[29] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 430–440, Piscataway, NJ, USA, 2012. IEEE Press.

[30] D. J. Gilmore. Models of debugging. *Acta Psychologica*, 78(1):151–172, 1991.

[31] S. Grant, J. R. Cordy, and D. Skillicorn. Automated concept location using independent component analysis. In *15th Working Conference on Reverse Engineering, 2008.*, pages 138–142, 2008.

[32] L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc. Tidier: an identifier splitting approach using speech recognition techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[33] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 35–44, Washington, DC, USA, 2010. IEEE Computer Society.

[34] S. Hens, M. Monperrus, and M. Mezini. Semi-automatically extracting faqs to improve accessibility of software development knowledge. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 793–803. IEEE Press, 2012.

[35] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, pages 232–242. IEEE Computer Society, 2009.

[36] H.-Y. Hsu, J. A. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 439–442. IEEE Computer Society, 2008.

[37] D. A. Hull and G. Grefenstette. A detailed analysis of english stemming algorithms. Technical report, Xerox Research and Technology, 1996.

[38] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of International Conference on Automated Software Engineering*, pages 184–193, 2007.

[39] J. A. Jones. *Semi-automatic fault localization*. PhD thesis, Georgia Institute of Technology, 2008.

[40] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 16–26. ACM, 2007.

[41] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.

[42] R. Krovetz. Viewing morphology as an inference process. In *Proceedings of the 16th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 191–202. ACM, 1993.

[43] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49:230–243, 2007.

[44] D. Lawrie, D. Binkley, and C. Morrell. Normalizing source code vocabulary. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 3–12. IEEE, 2010.

[45] S. Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987.

[46] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. volume 40, pages 15–26. ACM, 2005.

[47] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. volume 30, pages 286–295. ACM, 2005.

[48] J. B. Lovins. *Development of a stemming algorithm*. MIT Information Processing Group, Electronic Systems Laboratory, 1968.

[49] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9), 2010.

[50] N. Madani, L. Guerrouj, M. Di Penta, Y. Gueheneuc, and G. Antoniol. Recognizing words from source code identifiers using speech recognition techniques. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 68–77. IEEE, 2010.

[51] J. I. Maletic and A. Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Tools with Artificial Intelligence, 2000. ICTAI 2000. Proceedings. 12th IEEE International Conference on*, pages 46–53. IEEE, 2000.

[52] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of International Conference on Software Engineering*, 2001.

[53] A. Marcus and V. Rajlich. Panel: Identifications of concepts, features, and concerns in source code. 2005.

[54] W. Masri. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability*, 20(2):121–147, 2010.

[55] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[56] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.

[57] M. Ohba and K. Gondow. Toward mining concept keywords from identifiers in large software projects. 30(4):1–5, 2005.

[58] C. D. Paice. Another stemmer. *SIGIR Forum*, 24(3):56–61, Nov. 1990.

[59] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language api descriptions. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 815–825. IEEE Press, 2012.

[60] M. Perscheid, M. Haupt, R. Hirschfeld, and H. Masuhara. Test-driven Fault Navigation for Debugging Reproducible Failures. *Journal of the Japan Society for Software Science and Technology*, 29(3):3_188–3_211, 2012.

[61] M. F. Porter. An algorithm for suffix stripping. *Program: electronic library and information systems*, 14(3):130–137, 1980.

[62] M. F. Porter. Software reconnaissance: Mapping program feature to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[63] M. F. Porter. Snowball: A language for stemming algorithms, 2001.

[64] D. Poshyvanyk, M. Gethers, and A. Marcus. Concept location using formal concept analysis and information retrieval. *ACM Trans. Softw. Eng. Methodol.*, 21(4):23:1–23:34, Feb. 2013.

[65] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Softw. Eng.*, 33(6):420–432, June 2007.

[66] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 271–278. IEEE, 2002.

[67] M. Revelle and D. Poshyvanyk. An exploratory study on assessing feature location techniques. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 218–222. IEEE, 2009.

[68] P. Schone and D. Jurafsky. Knowledge-free induction of morphology using latent semantic analysis. In *Proceedings of the 2nd workshop on Learning language in logic and the 4th conference on Computational natural language learning-Volume 7*, pages 67–72. Association for Computational Linguistics, 2000.

[69] M. C. Scully. Augmenting program understanding with test case based methods. Technical report, Software Engineering Research Center, University of Florida, 1993.

[70] F. Servant and J. A. Jones. WhoseFault: Automatic Developer-to-Fault Assignment through Fault Localization. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[71] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of International Conference on Aspect-Oriented Software Development*, pages 212–224, 2007.

[72] B. Shneiderman. Measuring computer program quality and comprehension. *International Journal of Man-Machine Studies*, 9(4):465–478, 1977.

[73] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[74] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 71–80, 2011.

[75] M. Srivastav, Y. Singh, C. Gupta, and D. S. Chauhan. Complexity estimation approach for debugging in parallel. In *Computer Research and Development, 2010 Second International Conference on*, pages 223–227. IEEE, 2010.

[76] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* icomment: Bugs or bad comments? */. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 145–158. ACM, 2007.

[77] B. L. Vinz and L. H. Etzkorn. A synergistic approach to program comprehension. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 69–73. IEEE, 2006.

[78] A. von Mayrhauser and A. M. Vans. Program understanding behavior during debugging of large scale software. In *Papers presented at the seventh workshop on Empirical studies of programmers*, pages 157–179. ACM, 1997.

[79] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering*, pages 461–470. ACM, 2008.

[80] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, July 1982.

[81] A. Wiese, V. Ho, and E. Hill. A comparison of stemmers on source code identifiers for software search. In *Proceedings of the International Conference on Software Maintenance*, 2011.

[82] N. Wilde and C. Casey. Early field experience with the software reconnaissance technique for program comprehension. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 312–318, Washington, DC, USA, 1996. IEEE Computer Society.

[83] W. E. Wong, T. Wei, Y. Qi, and L. Zhao. A crosstab-based statistical method for effective fault localization. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 42–51. IEEE, 2008.

[84] C. Yilmaz, A. Paradkar, and C. Williams. Time will tell: fault localization using time spectra. In *Proceedings of International Conference on Software Engineering*, pages 81–90, 2008.

[85] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

[86] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. Sniafl: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(2):195–226, 2006.

[87] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112. ACM, 2006.

Appendix A

Appendix

Table A.1: First detailed experimental results for experiment five.

	Space Invaders			
	With Mantis		**Without Mantis**	
	Task Success	Task Failure	Task success	Task Failure
Task Completion Time (minutes)	12.12, 12.19, 24.10, 4.30, 2.18, 2.42, 11.14, 8.30, 0.39, 2.53, 8.0, 9.0, 6.42	25, 25, 25	12.37, 0.41, 3.32, 7.29, 12.46, 7.11, 21.16, 21.58, 6.34, 8.50, 7.28, 8.56, 8.42	25, 25, 25
Use of Concordance Prior to Start	1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1	1, 0, 1		
Time Using Concordance prior to Start (seconds)	26, 5, 13, 3, 42, 3	19, 1		
Time Using Mantis Prior to Start (seconds)	60, 38, 13, 0, 1, 0, 11, 42, 2, 0, 60, 60, 14	47, 3, 8		
Number of Mantis Uses During Debugging	3, 7, 11, 1, 0, 1, 2, 2, 0, 1, 5, 6, 1	7, 1, 3		
Time Duration of Mantis Use During Debugging (seconds)	[[5, 14, 29], [3, 3, 5, 2, 3, 10, 5, 12, 17, 2, 8, 31], [7, 9, 21, 1, 11, 9, 6, 4, 1, 5, 1], [18], [4], [13, 24], [8, 22], [6], [15, 10, 3, 2, 2], [10, 5, 10, 2, 1, 3], [2]]	[[8, 3, 21, 15, 10, 15, 11], [5], [1, 2, 24]]		
Use of Concordance During Debugging	1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1	1, 0, 1		
Found Bug	1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1	1, 1, 1, 1	1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1	0, 1, 0, 1
Years of Experience	f, d, c, c, c, d, c, c, c, c, e, d, f	c, c, d	e, e, c, c, c, b, c, d, d, f, c, d, d	d, e, c

Table A.2: Second detailed experimental results for experiment five.

	NanoXML			
	With Mantis		**Without Mantis**	
	Task Success	Task Failure	Task success	Task Failure
Task Completion Time (minutes)	22.03, 18.41, 16.45, 10.11	25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25	13.30	25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25
Use of Concordance Prior to Start	0, 0, 1, 1	0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0		
Time Using Concordance prior to Start (seconds)	15, 55	22, 14, 35, 38, 28, 12		
Time Using Mantis Prior to Start (seconds)	49, 8, 15, 60	73, 34, 26, 0, 90, 0, 35, 51, 28, 11, 8, 0		
Number of Mantis Uses During Debugging	13, 12, 5, 1	14, 7, 8, 15, 6, 8, 4, 7, 7, 11, 14, 4		
Time Duration of Mantis Use During Debugging (seconds)	[[25, 11, 10, 4, 8, 5, 1, 3, 7, 7, 1, 1, 1], [2, 2, 2, 5, 1, 2, 1, 2, 4, 3, 2, 1], [5, 5, 5, 16, 18], [16]]	[[73, 18, 2, 45, 17, 8, 32, 14, 16, 30, 2, 29, 21, 12], [11, 5, 1, 6, 23, 13, 13], [5, 4, 20, 2, 13, 28, 12, 5, 16], [13, 2, 4, 4, 2, 3, 9, 23, 48, 4, 4, 9, 11, 19, 9], [14, 12, 2, 8, 27, 5], [17], [30], [29], [5], [12], [29], [2], [23], [1, 3, 1, 4], [17, 19, 16, 72, 11, 2, 9], [6, 14, 9, 4, 14, 12, 26], [2, 24, 2, 12, 10, 9, 3, 6, 10, 4, 18], [8, 3, 8, 123, 21, 40, 59, 20, 2, 26, 13], [60, 2, 30, 2]]		
Use of Concordance During Debugging	1, 0, 1, 1	1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1		
Found Bug	1, 1, 1, 1	1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0	1	0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0
Years of Experience	f, f, c, e	d, f, , d, d, e, e, c, d, d, f, c, e	c	f, d, c, c, c, c, b, c, d, c, c, c, c