Automatically Describing Software Faults

Nicholas DiGiuseppe
Department of Informatics
University of California, Irvine
Irvine, CA, U.S.A. 92697-3440
nicholas.digiuseppe@uci.edu; www.ndigiuseppe.com

ABSTRACT
A developer’s ability to successfully debug a fault is directly related to their ability to comprehend the fault. Notwithstanding improvements in software-maintenance automation, this fault comprehension task remains largely manual and time consuming. I propose an automated approach to describe software faults, thus ameliorating comprehension and reducing manual effort. My approach leverages dynamic analysis, fault localization, and source-code mining to produce a succinct, natural-language fault summary.

Categories and Subject Descriptors
D.2.5 [Software]: Testing and Debugging

General Terms
Debugging Aid

Keywords
Fault Comprehension, Latent Semantic Analysis, Debugging, Automated Testing

1. RESEARCH PROBLEM
Current research suggests that software maintenance (including debugging) consumes up to 90% of industry budgets. To reduce this burden researchers have investigated the debugging process to potential debugging aids.

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” [13]. 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 [21]. Bettenburg et al. found that a quality bug description elucidates comprehension; consequently increasing the likelihood of the bug getting fixed [4].

In summary: before a fault can be fixed it must first be understood.

I propose Semantic Fault Diagnosis: an automated approach to generate natural-language fault descriptions that can ameliorate fault comprehension and reduce manual effort during debugging.

2. EXISTING APPROACHES
To the authors knowledge, Semantic Fault Diagnosis (SFD) is the first work of its kind to automatically generate fault descriptions. However, existing research can automatically provide information regarding a source code and faults. This section highlights three such fields: program comprehension, statistical fault-localization, and fault contextualization.

Program Comprehension
The most common approach related to this work is a body of research that automatically identifies features or components within the source code. Examples of this approach include Maletic and Marcus [19], Kuhn et al. [17], and Grant et al. [14]. Typically, these techniques use latent semantic indexing to reduce source code to modular-level topics (e.g., at the method level). These topics are then used to describe functionality of a given segment of code. These feature-extraction techniques are similar in their utilization of source-code text and summarizing code. However, my proposal diverges from these in its purpose, use of dynamic information, and ability to summarize disjoint segments of code.

Statistical Fault-Localization
My proposal is related to statistical fault-localization (SFL) in two ways: (1) my technique builds upon them, and (2) they describe an attribute of faults (i.e., their location). Lukins et al. [18] and Andrzejewski et al. [2] both produced SFL techniques that have meaningful similarities to my proposal. Lukins uses latent semantic analysis to reduce source code, developer comments, and text found in bug reports to topics. Then topic similarity between bug reports and the source code is computed to identify likely locations of the bug. Andrzejewski extends those topic models by partitioning them into "usage topics" and "failure topics." The insight leveraged by these techniques is that topics occurring frequently in the bug report and the source text are more likely the cause of the failure. Two significant dissimilarities between my proposal and these techniques is that: (1) these techniques require a bug report (i.e., an existing fault description) whereas SFD only requires execution profiles, and (2) these techniques describe a fault’s location whereas SFD describes a fault.
3. PROPOSED SOLUTION

I 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 preliminary knowledge of the bug or the failure.

The remainder of the paper is organized as follows: the expected contributions, methodology, current evaluation strategies, work to date, and the author’s publications.

3.1 Expected Contributions

The expected contributions of my dissertation are the ability to automatically generate natural-language fault descriptions, thereby facilitating debugging and reducing the required manual effort. The goals of my work are to develop a technique that: (1) fully automates the fault-description process, (2) leverages knowledge embedded in source code and developer comments, and (3) provides a simple, cognitive fault-description.

SFD will provide a short listing of keywords that succinctly describes a fault and its related functionality. In addition, because this approach presents keywords mined directly from source code, its results can form a basis for search queries during later debugging stages. These results can facilitate the acquisition of sufficient fault-comprehension, enabling developers to more quickly and easily debug the fault. Further, I will implement my methodology to enable experimentation through the generation of empirical data.

3.2 Methodology

My approach involves seven steps (as shown in Figure 1) that can all be completely automated. I describe this approach step by step in the following subsections (as previously described in [10]).

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.

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

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.”

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 stemming. In addition, common natural-language words, such as “the” and “of” can be removed. These words are often referred to as stop words.

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.

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, or the results may be visualized and correlated with the locations in the program.

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 cause of the errors.
3.3 Evaluation Strategy

To evaluate these fault descriptions, I propose four types of experiments: (1) a quantitative evaluation leveraging existing fault descriptions, (2) a qualitative evaluation via assessments from real developers, (3) users studies to evaluate the practical benefit SFD, and (4) an evaluation to determine the correlation between each step in the SFD process and the overall results.

Experiment one enables the use of large-scale subjects with years of development, and existing quality fault descriptions. To perform experiment one, 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). Precision, in information retrieval, is defined as the quantity of words in the query that also exist in the corpus, divided by the cardinality of the query. I suggest that precision be used (and not recall) because recall would be defined as the percentage of the words in the bug-report text that are part of the fault diagnosis. Bug reports often contain large amounts of text, at least relative to the few terms in the diagnoses, and most are merely required for their sentences to be linguistically correct. As such, precision scores are a good indicator of SFD quality, whereas recall is not.

While experiment one 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 two 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 one. To perform experiment two, 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 three enables the measurement of the practical benefit of SFD via a user study. For this experiment, I propose assessing developers as they debug a program where some developers have access to SFD results and others do not. I suggest measuring completion rate (i.e., did they fix the bug), time to completion, and their SFD usage patterns. This type of user study enables the measurement of SFD’s impact during the actual debugging process and can provide insights into its usage requirements for success.

Experiment four 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 with one of the previous methods as described in Experiment 1–3. This experiment measures the impact and correlation between the various sub-processes in SFD and the end result. This can enable an understanding when attempting to customize results, and when introducing improvements.

3.4 Work to Date

In previous work [10] I presented the motivation for this approach, along with the basic seven-step methodology (as presented in Section 3.2). In that work I also presented a prototype to generate fault descriptions and initial case studies. These initial case studies demonstrated that as a proof of concept SFD could provide semantically meaningful words to ameliorate fault comprehension without an extensive investigation into the source or having prior knowledge.

In addition to this preliminary work that defined the scope of the fault-comprehension problem and proposed my approach, I recently submitted a paper that describes a SFD implementation and initial evaluations. The built tool acts as a framework enabling a variety of SFD configurations (i.e., the use of different stemmers, splitters, and correlation algorithms that have been popular in past work along with two different result-presentation formats). The two result-
presentation formats are a tag-cloud-based visualization and ranked, bucketed word lists.

In addition to the implementation, this work also performs experiment one (as outlined in Section 3.3). For this experiment I used the program AspectJ, and over 50 real faults and bug reports. Initial results of experiment one were promising — a precision score averaging above 65%.

To address the limitations of experiment one I performed an investigation of the results to determine the quality of the words and found that (at least) anecdotally they were semantically valuable. Further, this anecdotal review found that many of the words which are not in the bug report are synonyms for words that do exist in the bug report.

### 4. AUTHOR’S PUBLICATIONS

In this section I present a listing of all my current publications. Due to space limitations, I cannot present a description for these publications and thus direct the reader to the individual papers.

As an undergraduate, I was an author on the following three papers: [1, 12, 15]. Since beginning my PhD, I have been an author on one workshop paper [5], two short papers [9, 10] and three long papers [7, 8, 11]. In addition to these publications, I am also an author on six papers currently under review.

### 5. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation, through Award CCF-1116943 and through Graduate Research Fellowship under Grant No. DGE-0808392. I also want to thank my advisor James A. Jones for his guidance.

### 6. REFERENCES