Semantic Fault Diagnosis: Automatic Natural-Language Fault Descriptions

Nicholas DiGiuseppe
University of California, Irvine
Department of Informatics
nicholas.digiuseppe@uci.edu

James A. Jones
University of California, Irvine
Department of Informatics
jajones@ics.uci.edu

ABSTRACT

Before a fault can be fixed, it first must be understood. However, understanding *why* a system fails is often a difficult and time consuming process. While current automated-debugging techniques provide assistance in knowing *where* a fault is, developers are left unaided in understanding *what* a fault is, and *why* the system is failing. We present *Semantic Fault Diagnosis* (SFD), a technique that leverages lexicographic and dynamic information to automatically capture natural-language fault descriptors. SFD utilizes class names, method names, variable expressions, developer comments, and keywords from the source code to describe a fault. SFD can be used immediately after observing a failing execution and requires no input from developers or bug reports. In addition we present motivating examples and results from a SFD prototype to serve as a proof of concept.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Diagnostics*

Keywords

Program Comprehension, Testing, Maintenance

1. INTRODUCTION

Software maintenance continues to consume a large portion of a project's total resources. This is in part because software has faults that can be expensive and time consuming to find and fix. One factor influencing this expense is expertise, or more simply put: before a fault can be fixed, it first must be understood. This paper presents *Semantic Fault Diagnosis* (SFD), a technique for generating automated natural-language fault descriptions.

One area of previous research has emphasized automatically locating a fault (*e.g.,* [8,9]). These techniques produce a ranked series of locations that are likely to contain the fault. Unfortunately, these techniques often assume that when developers see a fault they will immediately recognize it, understand why the code is broken, and understand how to fix it. Recent studies have suggested that this assumed perfect understanding is unlikely [10].

Concurrently, program-comprehension research has investigated feature-extraction techniques (*e.g.,* [4]). These techniques provide understanding for modular levels of source code (*e.g.,* the method level). Feature extraction techniques typically reduce a modular section of source code to a small quantity of representational topics. Unfortunately, these techniques often require user input (*e.g.,* a user query to locate a topic in the code), and often limit topic definitions to contiguous, modular boundaries.

The technique presented in this paper — *Semantic Fault Diagnosis* — merges the fundamental ideas of statistical fault localization and feature extraction in a novel way. Our objective is to facilitate debugging by presenting a developer with a description of their fault derived from the natural language of the source code. Our rationale is motivated in part by two studies. The first, from Bettenburg et al. [1] of over 150 software developers, concludes that a quality fault description would greatly assist practitioners in finding and fixing faults. The second, from Sayyad et al. [11], observes that "software engineers really need…'key' items of information, not a large amount of detail," to fix faults.

The fundamental idea behind using source code to generate fault descriptions is that source code is rich with conceptual information as to what the system *is* doing, and what it *should* to be doing. Biggerstaff et al. [2] examined the difference between the implementation reality and human intention. The source code provides the current structure and semantics of the program (*i.e.,* what the software *is*), and also includes syntax that describes semantic intent (*i.e.,* what the software *should be*). The semantic intent is often contained within source-code comments and variable and method identifiers that 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. Letha et al. [7] find that, "from the computer code *what* task is being done can be determined, but it is only from the comments that *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. More simply, developers require executable code and comments to gain sufficient understanding of what the code is doing and what it should do when modifying a software system in any nontrivial way.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1614-9/12/11 ...$15.00.

Therefore, we postulate that if a developer is given the appropriate terms and comments from the source code, they can understand and diagnose faults more quickly and simply.

The main contribution of this work is a technique for an automatic fault-description generator — Semantic Fault Diagnosis (SFD). SFD leverages executable source code and developer comments to produce a fault description immediately after execution failure. These descriptors are designed to elucidate the debugging process.

2. NOVEL FAULT DESCRIPTIONS

In previous automated debugging research there has been an emphasis on localizing a fault (*e.g.,* slicing-based or statistical fault localization). These techniques provide a developer with a list of locations in the code where the fault is likely to be. Unfortunately, data suggests that without contextual understanding, a developer will not be able to recognize the fault, nor find and fix it [10]. Other work has focused on providing control sequences (as opposed to a single location) to enable an understanding of a sequence of events that led to failure.

The novelty of SFD derives from its emphasis on the *why* of a failure by way of natural language as opposed to the *where* of a failure with structural, control-flow sequences. Our goal is to facilitate understanding regarding what the code is doing wrong, not provide locational data. However, we see SFD as complimentary to location-based techniques. We expect a developer to gain an understanding of their failure through SFD, then leverage location-based techniques to inform the fault's position.

In addition to its mechanics, SFD enables a new level of developer automation, which may positively influence the software-development process. To produce a semantically valuable fault description, SFD utilizes explanatory encoding information from the source code. Explanatory encoding ranges from programmer intentions that are encoded in source-code comments [7] to contextual understanding conveyed by naming conventions. SFD scans source code, processes the found terms, and correlates term use with failure. This process allows words that describe the fault to be automatically extracted and presented to the developer.

3. SEMANTIC FAULT DIAGNOSIS

Semantic Fault Diagnosis (SFD) is the process of automatically providing natural-language fault descriptions to developers based upon source code and dynamic information. We imagine SFD being employed in the software-development process immediately after testing failures are observed. This automated step can be configured to occur upon testing failure because no user input, outside data, nor bug reports are required. In such a scenario, the developer will be presented with not only the testing pass/fail results, but also a short list of words that can describe the topics involved in the fault and its failures. This list can guide the developer in knowing what to look for, thereby reducing their expertise burden. Additionally, because the words are all taken from the code, developers can search for their uses to improve localization. Furthermore, because descriptors are provided automatically and early, these words can be used to assist in writing bug reports and in assigning the debugging task to an appropriately skilled developer.

To create a semantic fault diagnosis, there are seven fully-automated steps as shown in Figure 1. SFD requires access to the source code, at least one passing and failing test case, a coverage-based instrumenter and a statistical fault-localization technique.

Step 1: Instrument Code. We instrument the program for statement coverage. This lightweight instrumentation can be achieved with existing tools like Cobertura[1] or Gcov.[2]

Step 2: Run Test Suite. We execute the instrumented source code over the selected test set, saving the pass/fail data and the coverage information produced as a result of the instrumentation. This test set can be of any size as long as it contains at least one passing and one failing test case.

Step 3: Perform Fault Localization. We use the execution coverage and the pass/fail data to perform fault-localization (FL). Any FL technique will suffice for this step, however techniques that utilize multiple failing runs and provide weighted scores for *all* executed lines are preferred.

Step 4: Parse Code. We parse the code to extract the terms. The parsing can be performed at a variety of levels of detail. For example, the method name `getReducedCost()`, could be parsed as a single term, "getReducedCost()," or split into its constituent natural-language terms: "get," "reduced," and "cost."

Step 5: Normalize Terms. We normalize the extracted terms from Step 4. Again, this step presents options for the degree to which the terms are processed: *e.g.,* stemming techniques can be employed to normalize verb tense, capitalization normalization can be performed, and synonym analysis can be performed to account for different terms that describe similar topics. For example, the term set {"waiting," "Waits," "paused"} could be normalized to {"wait," "pause"} or {"stop"}, or left unaltered altogether.

Step 6: Correlate Terms. Utilizing the fault-localization results to identify which locations in the source code are likely to be faulty and to what degree, we compute the fault-correlation for the individual terms that were extracted and processed in Steps 4 and 5. We compute a "fault-correlation" score for each term by computing the mean of the fault-correlation score that was given by the fault-localization technique for each of the locations that the term was found. For example, if the normalized word "wait" appears in three lines in the code and the FL technique gave those three lines the following scores $85, 51, 62$, then the word "wait" would gain an overall score of $66 = (85 + 51 + 62)/3$.

We also assign scores to developer comments and the terms therein. However, because they are not executed, comments need to be assigned suspicion scores based on the code they relate to. This can be done with a topical analysis (*e.g.,* attempting to discern which executed code the comments relate to) or more simply by assessing the proximity of the comments in the code with the executed code.

Step 7: Process Results. We present the top n terms and their scores to the developer. This presentation can be done in a variety of ways. For example, the scores can be presented in terms of: (1) their origin in the codebase, *e.g.,* method name, variable expression, or developer comment; (2) their use, *e.g.,* what lines they appear on; or (3) the top n words after sorting by score.

[1]http://cobertura.sourceforge.net/
[2]http://gcc.gnu.org/

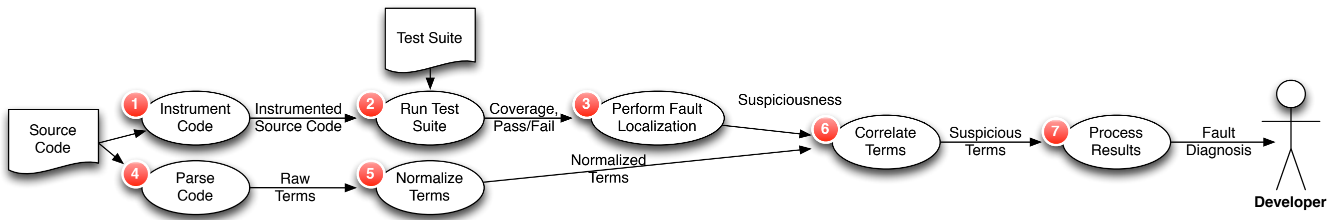

Figure 1: A process diagram depicting the seven steps for performing Semantic Fault Diagnosis.

4. DEMONSTRATION OF USE

In this section, we present two demonstrations of using a semantic fault-diagnosis technique on two real programs. The first example is from the classic game TETRIS[3] and the second is from an XML parser NANOXML, which can be downloaded from the Software-artifact Infrastructure Repository [6]. To provide these demonstrations, we implemented a prototype SFD tool.

4.1 Tetris

In the game TETRIS, the user guides falling blocks to create complete rows. To accomplish this, users can move falling blocks left and right. However, a fault exists in this version that causes the falling blocks to incorrectly move when the user presses the left arrow.

The following is the output of the top five terms produced using our prototype semantic fault-diagnosis tool. As can be seen, a quick reading of this diagnosis implies that the faulty topic is moving the figure to the left, which is exactly the nature of the fault and its failures.

Top Terms for Tetris
• KeyEvent.VK_LEFT • figure.moveLeft • moveLeft() • "Moves the figure one step to the left." • left

We also note that simply examining which test cases pass and which fail would not have provided this diagnostic information. For this fault, over 90% of the test cases for TETRIS failed — in fact, none of the test cases specifically tested the "moving left" functionality. Nevertheless, over 90% of the test cases exercised that functionality (and 10% did not), which was enough to allow our prototype implementation to identify that functionality as the determining factor that correlated with failure.

4.2 NanoXML

The NANOXML program is a tool that is used to parse XML data. The specification of NANOXML defines a flag "CDATA" that instructs the parser to ignore all text inside its block. Unfortunately, this version of NANOXML contains a fault that causes it to interpret and process the code within a "CDATA" block, rather than ignore it.

When we run our prototype semantic fault-diagnosis tool on this program and its test suite, we get the following output. As can be seen, the diagnosis indicates that the fault and its failures concern topics that include "content" and "PCData." And, while these terms are not specifically the term "CDATA" that would have been ideal, a developer of NANOXML would be aware that "PCData" stands for "processed CDATA," which indeed is indicative of the faulty issue. Moreover, the method `setContent()` is where interpreted and processed data is stored, and for this fault, the processed CDATA is incorrectly being processed as content.

Top Terms for NanoXML
• elt.setContent(str • setContent() • addPCData() • Indicates that a new #PCData element has been encountered. • PCDataAdded

Much like the TETRIS demonstration, we also note that simply examining which test cases passed and which failed would not have provided this diagnostic information. For this fault, over 60% of the test cases for NANOXML failed — in fact, none of the test cases specifically tested the "ignore CDATA" functionality.

5. RELATED WORK

This work is primarily related to two bodies of research: statistical fault localization and feature extraction. Because these fields are mostly disjoint, we separately discuss each, along with how SFD relates to the authors' previous work.

5.1 Statistical Fault Localization

Lukins et al. [9] used latent-Dirichlet allocation (LDA) on source code to localize a fault. Their technique uses LDA on developer comments, method names, and other textual references to generate topics for modular locations of code (*e.g.,* methods). Next, their algorithm uses LDA on a bug report to generate topics, then compares the two topic sets. The locations representing a high degree of topical similarity are presented to developers as the likely location of the fault.

Lukins' work is related to semantic fault diagnosis in that they both leverage lexical and semantic analysis of the source code. However, they differ in their manipulation of lexical elements and in purpose. SFD requires no developer input and uses dynamic information to *describe* a fault while the approach by Lukins et al. needs a developer's description (from a bug report), is static, and attempts to *locate* a fault.

5.2 Feature Extraction

Dit et al. [5] presented a survey of feature-extraction techniques. Of relevance here, they provide three observations: (1) feature-extraction techniques use three primary input types (dynamic, static, and textual); (2) most feature-extraction techniques define features for contiguous, discrete modules (*e.g.,* methods, classes, or packages); and (3) many

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

feature-extraction techniques use topic-modeling techniques to represent features.

SFD demonstrates certain similarities to the feature-extraction community. For example, both techniques strive for increased developer comprehension by providing essential information through the use of abstraction. However, the main difference between these techniques is that feature extraction explains the system as a whole or discrete modules based upon code structure, whereas SFD targets a fault irrespective of code structure. At a high level, SFD can be considered a unique type of feature extraction that explains fault-to-concept correlation as opposed to module explanations through code-structure-based explanations.

5.3 Author's Related Works

This work is an extension of the author's previous work on fault localization and comprehension. Jones et al. [8] presented TARANTULA, which uses dynamic information to correlate instructions with passing and failing events, thereby estimating the likelihood that an instruction is faulty. More recently, DiGiuseppe and Jones investigated faults' ability to alter software behavior [3] by investigating how faults interact within an execution and the likelihood of those events.

SFD builds directly on these two ideas. As can be seen in Section 3, SFD directly depends on statistical fault localization to approximate the correlation of source-code concepts and failure. Further, by not being constrained to providing topics at the modular level, but instead allowing concepts to be mined across modules, SFD can provide finer grained, and thus likely more descriptive, fault concepts.

6. EXPECTED FEEDBACK

SFD provides descriptions of software faults. This description could be presented in a variety of ways and include or exclude various software artifacts. For example, SFD results can be presented: in terms of where words originated from (*e.g.,* method names, variable expressions, or developer comments) including links to the source code, as a simple ranked list in isolation, as a central theme (*i.e.,* use latent semantic analysis to deduce a topic from the words) matched to test cases containing very similar topics, or some alternative variation. We are interested in discussing how developers should receive this information, the impact of those choices, and potential trade-offs.

Additionally, when considering usefulness, we are interested in discussing evaluation strategies for SFD. While comparisons could be made from words generated by SFD to descriptions given by actual developers (in bug reports, or commit messages of a repository) that may fail to demonstrate the words that contain meaningful value to developers. We are interested in discussing quantitative evaluations that measure "value."

We are also interested in discussing how the value of words changes based upon their origin. For example, words selected from comments, class names or variable expressions might best describe faults. However, it could be that only when supplementary artifacts are added (*e.g.,* test-case descriptions or documentation) that SFD's word choices are meaningful to developers. Discussing the ramifications of identifying semantic value by source code "demographics" could enlighten the community regarding design (what articles are of increased value to maintenance) and naming conventions (what naming styles make maintenance easier).

7. CONCLUSIONS

In this paper we present a technique for automatically generating words that describe a fault — *semantic fault diagnosis.* We present demonstrations that use a functional prototype, and we outline potential directions for future work. SFD utilizes all source-code text (*e.g.,* variables expressions, method names, and developer comments) and statistical fault localization to approximate the words that best describe a fault.

SFD requires no input from developers and can be utilized immediately after witnessing testing or execution failures. This immediacy allows for a variety of uses in the software lifecycle ranging from debugging assistance to assisted bug report generation.

8. 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.

9. REFERENCES

[1] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *International Symposium on Foundations of Software Engineering*, 2008.

[2] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *International Conference on Software Engineering*, 1993.

[3] N. DiGiuseppe and J. A. Jones. Fault interaction and its repercussions. In *International Conference on Software Maintenance*, 2011.

[4] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol. Can better identifier splitting techniques help feature location? In *International Conference on Program Comprehension*, 2011.

[5] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[6] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 2005.

[7] L. H. Etzkorn, C. G. Davis, and L. L. Bowen. The language of comments in computer software: A sublanguage of english. *Journal of Pragmatics*, 2001.

[8] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering*, 2002.

[9] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Inf. Softw. Technol.*, 2010.

[10] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis*, 2011.

[11] J. Sayyad-Shirabad, T. C. Lethbridge, and S. Lyon. A little knowledge can go a long way towards program understanding. In *International Workshop on Program Comprehension*, 1997.