Are Automated Debugging Techniques Actually Helping Programmers?

Chris Parnin and Alessandro Orso
Georgia Institute of Technology
College of Computing
{chris.parnin|orso}@gatech.edu

ABSTRACT

Debugging is notoriously difficult and extremely time consuming. Researchers have therefore invested a considerable amount of effort in developing automated techniques and tools for supporting various debugging tasks. Although potentially useful, most of these techniques have yet to demonstrate their practical effectiveness. One common limitation of existing approaches, for instance, is their reliance on a set of strong assumptions on how developers behave when debugging (*e.g.,* the fact that examining a faulty statement in isolation is enough for a developer to understand and fix the corresponding bug). In more general terms, most existing techniques just focus on selecting subsets of potentially faulty statements and ranking them according to some criterion. By doing so, they ignore the fact that understanding the root cause of a failure typically involves complex activities, such as navigating program dependencies and rerunning the program with different inputs. The overall goal of this research is to investigate how developers use and benefit from automated debugging tools through a set of human studies. As a first step in this direction, we perform a preliminary study on a set of developers by providing them with an automated debugging tool and two tasks to be performed with and without the tool. Our results provide initial evidence that several assumptions made by automated debugging techniques do not hold in practice. Through an analysis of the results, we also provide insights on potential directions for future work in the area of automated debugging.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging Aids*

General Terms: Experimentation

Keywords: Statistical debugging, user studies

1. INTRODUCTION

When a software failure occurs, developers who want to eliminate the failure must perform three main activities. The first activity, *fault localization*, consists of identifying the program statement(s) responsible for the failure. The second activity, *fault understanding*, involves understanding the root cause of the failure. Finally, *fault correction* is determining how to modify the code to remove such root cause. Fault localization, understanding, and correction are referred to collectively with the term *debugging.*

Debugging is often a frustrating and time-consuming experience that can be responsible for a significant part of the cost of software maintenance [25]. This is especially true for today's software, whose complexity, configurability, portability, and dynamism exacerbate debugging challenges. For this reason, the idea of reducing the costs of debugging tasks through techniques that can improve efficiency and effectiveness of such tasks is ever compelling. In fact, in the last few years, there has been a great number of research techniques that support automating or semi-automating several debugging activities (*e.g.,* [1,3,8,11,21,29–31]). Collectively, these techniques have pushed forward the state of the art in debugging. However, there are several challenges in scaling and transitioning these techniques that must be addressed before the techniques are placed in the hands of developers.

In particular, one common issue with most existing approaches is that they tend to assume *perfect bug understanding*, that is, they assume that simply examining a faulty statement in isolation is always enough for a developer to detect, understand, and correct the corresponding bug. This simplistic view of the debugging process can be compelling, as it allows for collecting some objective information on the effectiveness of a debugging technique and provides a common ground for comparing alternative techniques. However, this view has now become a de-facto metric for guiding the definition of debugging techniques and assessing their usefulness, which is less than ideal. In fact, recently we are witnessing a situation where many researchers are just focusing on giving a faulty statement the highest rank (in some cases, with little gain over the state of the art).

Just focusing on statement selection and ranking ignores the fact that understanding the root cause of a failure typically involves complex activities (*e.g.,* navigating data and control dependences in the code, examining parts of the program state, rerunning the program with different inputs). We believe that, given the maturity of the field, it is now time to take into account the inherent complexity of these debugging activities in both the definition and, especially, the evaluation of debugging techniques. Such an evaluation should involve studies on how real developers use existing techniques and whether such use is actually beneficial.

Unfortunately, in 30 years since Weiser's foundational paper on program slicing [26], only a handful of empirical

ISSTA'11, July 17–21, 2011, Toronto, ON, Canada
Copyright 2011 ACM 978-1-4503-0562-4/11/05 ...$10.00

studies have evaluated how developers use debugging tools (*e.g.,* [6, 15, 16, 26, 28]). Even worse, the total number of participants in these studies amounts to less than 100 developers, and most programs studied are trivial, often involving less than 100 lines of code. As Francel and Rugaber noted [6], for instance, only 3 out of 111 papers on slicing based debugging techniques have considered issues with the use of the techniques in practice. Similar considerations could be made today about spectra-based debugging techniques (*e.g.,* [11, 17]).

The overall goal of this research is to address this gap in our understanding of automated debugging through a set of human studies that investigate what makes debugging aids work (or not work) in practice. Note that our goal is not to conclusively prove that a particular approach is better than another one; it is rather to gather insight on how to build better debugging tools and identify promising research directions in this area. As a first step towards our goal, in this paper we perform a preliminary study in which we investigate whether, and to what extent, a popular automated debugging technique [11] helps developers in their debugging tasks. More precisely, we selected a set of 34 developers with different degrees of expertise, assigned them two different debugging tasks, and compared their performance when using a representative automated debugging tool [11] and when using standard debugging tools available within the Eclipse IDE (`http://www.eclipse.org/`). In the study, we did not just examine whether developers can find bugs faster using an automated technique; we also tried to justify our results by analyzing in detail and understanding the developers' behavior when using debugging tools.

The study produced several interesting results. For example, we found that the use of an automated tool helped more experienced developers find faults faster in the case of an easy debugging task, but the same developers received no benefit from the use of the tool on a harder task. We also found that most developers, when provided with a list of ranked statements, do not examine the statements in the order provided, but rather search through the list based on some intuition on the nature of the fault (which limits the usefulness of pure statement ranking). In general, although the results of our study are still preliminary, they provide insight into the behavior of developers during debugging and strong evidence that several of the assumptions on which many automated debugging techniques rely do not actually hold in practice.

In this paper, we make the following contributions:

- A review of the main empirical studies on automated debugging tools presented in the literature.
- Two studies that investigate whether and to what extent debugging tools help developers.
- An analysis of the study results and a discussion of the implications of such results for future research in the area of fault localization and debugging in general.

2. DEBUGGING TECHNIQUES AND RELATED STUDIES

2.1 Automated Debugging Techniques

Over the years, researchers have defined increasingly sophisticated debugging approaches, going from mostly manual techniques to highly automated ones. Simultaneously, infrastructure to support debugging tasks has also been developed. As a result, the body of related work is broad. Here, we focus on representative examples and on the work that is most related to our proposed research.

Almost 30 years ago, Weiser proposed one of the first techniques for supporting automated debugging and, in particular, fault localization: program slicing [26, 27]. Given a program P and a variable v used at a statement s in P, slicing computes all of the statements in P that may affect the value of v at s. By definition, if the value of v in s is erroneous, then the faulty statements that led to such erroneous value must be in the slice. In other words, any statement that is not in the slice can be safely ignored during debugging. Although slicing can generate sets of relevant statements, in most realistic cases these sets are too large to be useful for debugging [30]. To address this issue, researchers investigated different variations of slicing aimed at reducing the size of the computed slices. In particular, Korel and Laski introduced dynamic slicing, which computes slices for a particular execution (*i.e.,* a statement is in the slice only if it may affect the value of v in s for the specific execution considered). In subsequent years, different variations of dynamic slicing have been proposed in the context of debugging, such as critical slices [4], relevant slices [9], data-flow slices [31], and pruned slices [30]. These techniques can considerably reduce the size of slices, and thus potentially improve debugging. However, the sets of relevant statements identified are often still fairly large, and slicing-based debugging techniques are rarely used in practice.

An alternative family of debugging techniques aim to overcome the limitations of slicing-based approaches by following a different philosophy. These techniques identify potentially faulty code by observing the characteristics of failing program executions, and often comparing them to the characteristics of passing executions. These approaches differ from one another in the type of information they use to characterize executions and statements—path profiles [22], counterexamples (*e.g.,* [1, 8] and approaches based on model checking in general), statement coverage [11], and predicate values [17, 19]—and in the specific type of mining performed on such information. Additional work in this area has investigated the use of clustering techniques to eliminate redundant executions and facilitate fault localization [10, 13, 18, 21].

One problem with this general class of techniques is that they focus exclusively on trying to reduce the number of statements developers need to examine when investigating a failure, under the assumption that examining a faulty statement in isolation is enough for a developer to detect and fix the corresponding bug. Unfortunately, it is unclear whether developers can actually determine the faulty nature of a statement by simply looking at it, without any additional information (*e.g.,* the state of the program when the statement was executed or the statements that were executed before or after that one). When using these techniques, this type of information can only be gathered by rerunning the program against the input that caused the failure being investigated. In general, without a clear understanding of how developers would use these techniques in practice, the potential effectiveness of such techniques remains unknown.

2.2 Empirical Studies with Programmers

As we mentioned in the Introduction, to the best of our knowledge only a handful of researchers have performed studies to evaluate automated debugging techniques and tools with programmers. This scarcity of studies limits our understanding of the effectiveness of these techniques, their practicality, and the validity of the assumptions on which they rely. In the rest of this section, we summarize the main empirical studies performed so far in this area.

In Weiser's initial study [26], 21 programmers examined 3 programs whose size ranged between 75 and 150 lines of code. The study did not directly evaluate if programmers could more effectively debug with slicing. Instead, programmers were (1) asked to debug a never-seen-before program P, (2) presented with different fragments of P of two types, slices and "random" snippets, and (3) requested to recognize the various code fragments. Overall, slices were recognized significantly more often than other fragments, which suggests that programmers tend to follow the flow of execution when investigating a failure during debugging.

The first study to actually examine whether programmers with a slicing tool could debug more effectively was performed by Weiser and Lyle, but could not find any benefit [28]. In the study, they did not observe any improvement when developers debugged a small faulty program (100 LOC) using a slicing tool. In a follow-up study, they changed several parameters of the experiment. First, they used a smaller program (25 LOC). Second, instead of an interactive tool, they used a paper printout of the sliced program. Finally, they used a different slicing technique, called dicing, that combines slices from failing and passing test cases to reduce the number of statements to be examined. The results of this second study were positive, in that programmers provided with the printout of the slices were faster in finding some of the bugs than the programmers without that information.

More recent studies have examined the debugging behavior of programmers in relation to slicing. In an experiment with 17 students, Francel and Rugaber found that only four of those programmers were instinctively slicing [6]. The study also showed that these four developers had a better understanding of the program, in comparison to non-slicers, when debugging a 200 line program. Other differences were found between the groups, such as the fact that non-slicers were less careful and less systematic in their inspection.

In a subsequent study, Kusumoto and colleagues selected six students, provided three of them with a slicing tool, and assessed which group performed better during debugging [16]. The study involved finding bugs in 3 small programs (around 400 LOC) containing a total of 9 faults. No significant difference could be found between the groups in this case, so the researchers performed a simplified version of the experiment in which they used 6 smaller programs (25–52 LOC) and 6 faults. The results of this second study showed a significant difference in the amount of time needed by the two groups of students to locate the faults for some of the programs, but not for all of them.

Sherwood and Murphy presented an experiment in which six subjects were provided with an Eclipse plugin that showed differential coverage between execution traces (*e.g.,* a passing and a failing one) and used it to understand and fix faulty code on two real-world medium-sized programs [23]. Although no definitive conclusions could be drawn from the study, there was some evidence that novice programmers that used the tool could be more successful than experts who were not using it.

The most extensive evaluation of a debugging technique to date is the empirical study of the Whyline tool [14]. Whyline focuses on assisting novice users with formulating hypotheses and asking intuitive questions about a program's behavior. For example, a user can click on part of a program's interface and select the question "why is this rectangle blue?" The user would then be shown the statements responsible for that behavior, could ask further questions on those statements, and could continue the investigation in this interactive way. Merging visualization, dynamic slicing, automatic reasoning, and a slick user interface into one tool, Whyline shows what a mature program slicing tool can achieve; in a study with 20 subjects investigating two real bugs from ArgoUML (150 KLOC), participants that used Whyline were able to complete the task twice as fast than participants using only a traditional debugger [15].

In summary, as the brief survey in this section shows, empirical evidence of the usefulness of many automated debugging techniques is limited, in the case of slicing, when not completely absent, for most other types of techniques. This situation makes it difficult to assess the practical effectiveness of the techniques proposed in the literature and to understand which characteristics of a technique can make it successful. In this paper, we try to fill this gap by studying a set of developers while debugging real bugs with and without the help of an automated debugging technique. The rest of the paper discusses our study and its results.

3. RESEARCH QUESTIONS AND HYPOTHESES

3.1 Hypotheses

Intuitively, developers using an automated debugging tool should outperform developers that do not use the tool. Our first hypothesis is therefore as follows.

Hypothesis 1 - Programmers who debug with the assistance of automated debugging tools will locate bugs faster than programmers who debug code completely by hand.

We would also expect that an automated tool should be more useful for a more difficult debugging task. In this case, the tool should give developers an *edge* over traditional debuggers. We hence formulate a second hypothesis.

Hypothesis 2 - The effectiveness of an automated tool *increases* with the level of difficulty of the debugging task.

Finally, when considering the class of debugging techniques based on statement ranking, the central assumption of many previous evaluations is that a high rank should be correlated with less effort on the programmer's side because fewer statements must be inspected. Based on this assumption, we would expect the rank of the faulty statement to affect a programmer's performance. This concept is expressed by our third hypothesis.

Hypothesis 3 - The effectiveness of debugging when using a ranking based automated tool is affected by the rank of the faulty statement.

3.2 Research Questions

In our study, we also formulate several research questions concerning how developers use automated debugging tools in practice. Our first research question asks how realistic is the assumption that programmers would use a ranked list of statements provided by a tool. The rationale for this question is that the premise of most evaluations of debugging techniques is that developers investigate statements individually, one at a time, until they find the bug. This assumption gives a one-to-one mapping between the rank of the faulty statements and the assumed effectiveness of an algorithm.

Research Question 1 - How do developers navigate a list of statements ranked by suspiciousness? Do they visit them in order of suspiciousness or go from one statement to the other by following a different order?

Our second research question investigates if a programmer can identify a faulty statement just by looking at it. An implicit assumption in the model used by existing empirical evaluations is in fact that such *perfect bug understanding* exists. We want to assess how realistic this assumption is.

Research Question 2 - Does perfect bug understanding exist? How much effort is actually involved in inspecting and assessing potentially faulty statements?

Our final research question seeks to capture and understand the challenges, in terms of problems and also opportunities, faced by developers when using automated debugging tools.

Research Question 3 - What are the challenges involved in using automated debugging tools? What issues or barriers prevent their effective use? Can unexpected, emerging strategies be observed?

4. EXPERIMENTAL PROTOCOL

4.1 Program Subjects

In our experiment, we wanted to select program subjects that were similar to the ones used in previous studies and yet not trivial.

4.1.1 Tetris

As our first subject, we selected a Java implementation of Tetris (`http://www.percederberg.net/games/tetris/tetris-1.2-src.zip`). The program consists of 2,403 LOC including comments and blanks. From our previous experience in running programming experiments [20], popular games are ideal subjects, as participants are typically familiar with the behavior of the games and can readily identify game concepts in the source code.

4.1.2 NanoXML

As our second subject, we selected NanoXML, a library for XML parsing available from the SIR repository [5]. The version we used consists of 4,408 LOC including comments and blanks. NanoXML, besides being one of the largest subjects used in evaluations of automated debugging techniques to date, presents many characteristics that Tetris does not have. In particular, parsing is a domain with which many programmers are not familiar, especially for what concerns more advanced features of XML, such as namespaces and schemas. In this sense, using these two subjects lets us investigate two complementary situations—one where the users are somehow familiar with the code, and the other where they are not.

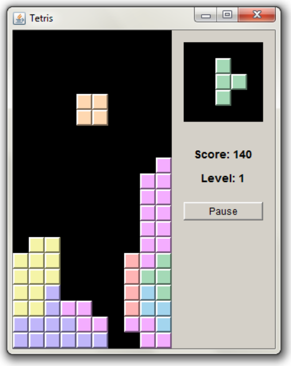


Figure 1: Tetris Task: Identify and fix the cause of the abnormal rotation of squares in Tetris.

4.2 Participants

We selected participants from the set of graduate students enrolled in graduate-level software engineering courses at Georgia Tech. As part of their coursework, these students have been instructed on debugging, including automated fault-localization techniques. Overall, we had a total of 34 participants, whose backgrounds represented a full range of experiences; several participants worked in industry for one or more years (or even ran their own startup companies) and returned to school, others did one or more internships in companies, and yet others had limited programming experiences outside of school. As every programmer must perform some debugging, from new hires to seasoned experts, we were comfortable with this variance in experiences, which could allow us to assess how debugging tools could help different types of programmers in different ways.

4.3 Tasks

We gave participants two main tasks, each involving the debugging of a failure in one of the two subjects considered. For each task, the participants were given a description of the failure and a way of reproducing it, and were asked to identify the fault(s) causing such failure and propose a fix.

4.3.1 Task 1

The first task involved identifying and fixing a fault in Tetris. The description of the failure consisted of the screenshot shown in Figure 1 and the following textual description:

> The rotation of a square block causes unusual behavior: The square block will rise upwards instead of rotating in place (which would have no observable effect). If needed, you can easily reproduce the failure by running Tetris and trying to rotate a square block.

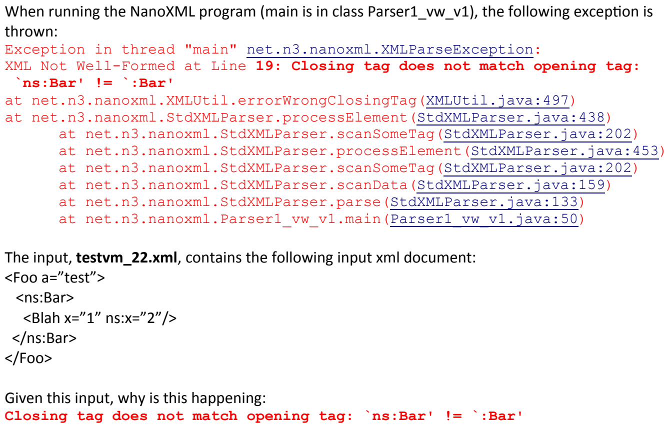
4.3.2 Task 2

The second task involved identifying and fixing a fault in NanoXML. The failure description, shown in Figure 2, contained a stack trace and a test input causing the failure. As the figure shows, the failure consisted of an XMLParseException that was raised because starting and closing XML tags did not match: “ns:Bar” != “:Bar”.

4.4 Instruments

4.4.1 Automated Debugging Technique

As a representative of current approaches to automated fault localization, we chose to use the Tarantula technique by



When running the NanoXML program (main is in class Parser1_vw_v1), the following exception is thrown:

```
Exception in thread "main" net.n3.nanoxml.XMLParseException:
XML Not Well-Formed at Line 19: Closing tag does not match opening tag:
 `ns:Bar' != `:Bar'
at net.n3.nanoxml.XMLUtil.errorWrongClosingTag(XMLUtil.java:497)
at net.n3.nanoxml.StdXMLParser.processElement(StdXMLParser.java:438)
      at net.n3.nanoxml.StdXMLParser.scanSomeTag(StdXMLParser.java:202)
      at net.n3.nanoxml.StdXMLParser.processElement(StdXMLParser.java:453)
      at net.n3.nanoxml.StdXMLParser.scanSomeTag(StdXMLParser.java:202)
      at net.n3.nanoxml.StdXMLParser.scanData(StdXMLParser.java:159)
      at net.n3.nanoxml.StdXMLParser.parse(StdXMLParser.java:133)
      at net.n3.nanoxml.Parser1_vw_v1.main(Parser1_vw_v1.java:50)
```

The input, **testvm_22.xml**, contains the following input xml document:

```
<Foo a=”test”>
  <ns:Bar>
   <Blah x=”1” ns:x=”2”/>
 </ns:Bar>
</Foo>
```

Given this input, why is this happening:

```
Closing tag does not match opening tag: `ns:Bar' != `:Bar'
```

Figure 2: NanoXML Task: Identify the cause of the failure in NanoXML and fix the problem.

Jones and colleagues [11], for several reasons. First, Tarantula is, like most state-of-the-art debugging techniques, based on some form of statistical ranking of potentially faulty statements. Second, a thorough empirical evaluation of Tarantula has shown that it can outperform other techniques [12]. (More recently, more techniques have been proposed, but their improvements, when present, are for the most part marginal and dependent on the context.) Finally, Tarantula is easy to explain and teach to developers.

4.4.2 Plugin

To make it easy for the participants to use the selected statistical debugging technique, we created an Eclipse plugin that provides the users with the ranked linked of statements that would be produced by Tarantula. We decided to keep the plugin's interface as simple as possible: a list of statements, ordered by suspiciousness, where clicking on a statement in the list opens the corresponding source file in Eclipse and navigates to that line of code. We believe that this approach has the twofold advantage of (1) letting us investigate our research questions directly, by having the participants operate on a ranked list of statements, and (2) clearly separating the benefits provided by the ranking based approach from those provided by the use of a more sophisticated interface, such as Tarantula's visualization [11].

The plugin, shown in Figure 3, works as follows. First, the user inputs a configuration file for a task by pressing the load file icon. Once the file is loaded, the plugin displays a table with several rows, where each rows shows a statement and the corresponding file name, line number, and suspiciousness score. Besides clicking on a statement to jump to it, as discussed above, users can also use a previous and next button to navigate through the statements.

To compute the ranked list of statements for the plugin, we used the Tarantula formulas provided in Reference [11], which require coverage data and pass/fail information for a set of test inputs. For both Tetris and NanoXML, we collected coverage data using Emma (`http://www.eclemma.org/`). For NanoXML, we used the test cases and pass/fail status for such test cases available from the SIR repository. For Tetris, for which no test cases were available, we wrote a capture-replay system that could record the keys pressed when playing Tetris and replay them as test cases. Overall, we collected 10 game sessions, 2 of which executed the faulty statement (*i.e.,* rotated a square block).

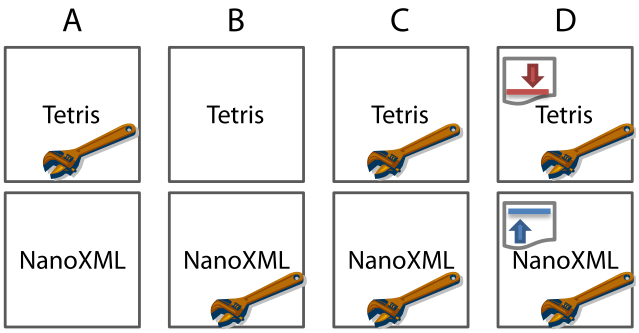


Figure 4: Participants are split into different groups having different conditions. Each box represents a task: the label in the box indicates the software subject for the task; the presence of a wrench indicates the use of the automated debugging tool for that task; the icons representing an arrow indicate tasks for which the rank of the faulty statement has been increased (up) or decreased (down).

4.4.3 Data Availability

Our Eclipse plugin, program subjects, and instructions sheets are available for researchers wishing to replicate our study at `http://www.cc.gatech.edu/˜vector/study/`.

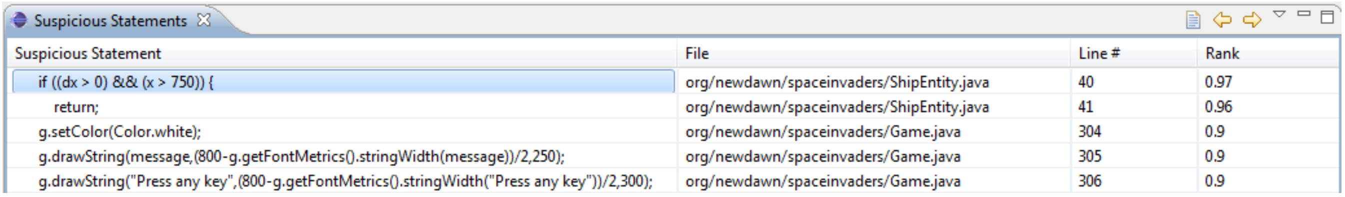
4.5 Method

To evaluate our Hypothesis 1, and assess whether participants could complete tasks *faster* when using an automated debugging tool (tool, hereafter), we created two experimental groups: A and B. Participants in group A were instructed to use the tool to solve the Tetris task. Conversely, participants in group B had to complete the Tetris task using only traditional debugging capabilities available within Eclipse. If the tool were effective, there should be a significant difference between the two group's task completion time.

We investigated our Hypothesis 2, and assessed whether participants *benefited* more from using the tool on harder tasks, by giving the experimental groups a second task: fix a fault in NanoXML. In group A, participants were limited to use only traditional debugging techniques available within Eclipse, whereas in group B, participants could also use the tool to solve the task. In this case, we compared the difference in performance for the groups using the tool for the Tetris and the NanoXML tasks. If the tool were more effective for harder tasks, the performance gain of participants using the tool for the NanoXML task should be better than that of participants using the tool on the Tetris task.

Our Hypothesis 3 aims to understand the effects of the *rank* of the faulty statement on task performance. To investigate this hypothesis, we created two new experimental groups: C and D. Both groups were given both the Tetris and the NanoXML tasks and were instructed to use the tool to complete the tasks. The difference between the two groups was that, for group D, we lowered the rank of the faulty statement for Tetris (*i.e.,* moved it down the list) and increased the rank of the faulty statement for NanoXML (*i.e.,* moved it up the list). If rank were an important factor, there should be a decrease in performance for the Tetris task and an increase in performance for the NanoXML task for group D.

A summary of the method we used for investigating our hypotheses can be seen in Figure 4.



Suspicious Statement	File	Line #	Rank
if ((dx > 0) && (x > 750)) {	org/newdawn/spaceinvaders/ShipEntity.java	40	0.97
return;	org/newdawn/spaceinvaders/ShipEntity.java	41	0.96
g.setColor(Color.white);	org/newdawn/spaceinvaders/Game.java	304	0.9
g.drawString(message,(800-g.getFontMetrics().stringWidth(message))/2,250);	org/newdawn/spaceinvaders/Game.java	305	0.9
g.drawString("Press any key",(800-g.getFontMetrics().stringWidth("Press any key"))/2,300);	org/newdawn/spaceinvaders/Game.java	306	0.9

Figure 3: Eclipse plugin for automated debugging based on Tarantula's ranking.

To answer Research Question 1 on how programmers use the ranked list of statements provided by the tool, we recorded a log of the navigation history of the participants that used the tool. To answer Research Question 2 on perfect bug understanding, we analyzed the log history to measure the time between clicking on the faulty statement and completing the task. Finally, to answer Research Question 3, we gave the participants a questionnaire that asked them to describe how they used the tool and report any issues and experiences they had with it.

4.6 Procedure

Participants performed the study in either a classroom or our lab. A week before the experiment, we gave the participants a chance to install the plugin and test it on a sample program. At the start of the experiment, the participants were instructed on the general purpose of the study and were told that they had a total of 30 minutes to complete each task, after which they should move to the next task. In total, we allocated 2 hours for the entire study: 30 minutes for instruction and setup, 1 hour for the tasks, and 30 minutes for completing a questionnaire at the end of the study.

To let the participants familiarize with the failures, we had them first replicate such failures. To measure task completion time, we instructed participants to record as their starting time the time when they begun looking at the code and investigating the failure. To measure the correctness of a solution and gain a better understanding of how participants found the fault, participants were asked to describe the reason for the failure and how they used the tool to find it. Once participants were done with their tasks, they completed the questionnaire about their use of the tool and experience with it.

5. STUDY RESULTS

We ran an initial experiment with 34 students split between groups A and B (see Section 4.5). Interestingly, we were able to find support for Hypothesis 1, but a flaw in the ranking for NanoXML left us unable to answer our research questions. We used this initial experiment to improve our methods and procedures for conducting the experiment.

In the following, we report the results of our main experiment, which was run with 24 students split between groups A and B. We also report the results of a follow-up experiment with an additional 10 students split between groups C and D (see Section 4.5).

We show the overall results for this part of the study in Table 1 and discuss the significance of these results in terms of our hypotheses in the following subsections.

5.1 Experts are Faster when Using the Tool

Table 1: Successful task completion time in minutes and seconds for all conditions.

Group	Tetris	NanoXML
A	8:51	22:30
B	14:28	20:16
C	10:12	15:12
D	12:36	18:30

For the Tetris task, the average completion time, in minutes and seconds, was 12:42 for group A (tool) and 14:14 for group B (traditional debugging). This difference is not statistically significant by a two-tailed t-test. However, as we noted above, we did observe a significant difference in a previous experiment comparing these conditions. For the NanoXML task, the average completion time, in minutes and seconds, was 20:16 for group B (tool) and 22:30 for group A (traditional debugging). These values were also not significantly different.

Analyzing the results in more detail, we observed that some participants received a much higher benefit from using the tool than others. We therefore split the participants from groups A and B into three groups based on their performance: low, medium, and high performers. The low performers were likely novices, as they were not able to complete any of the tasks within 30 minutes. The medium performers were able to complete at least one of the tasks (most often Tetris), and the high performers could complete both tasks.

Comparing the high performers (N=10) in groups A and B, the average completion time, in minutes and seconds, was 8:51 (A) versus 14:28 (B) for Tetris. This difference is statistically significantly ($p < 0.05$) by a two-tailed t-test.

Overall, we found **support for Hypothesis 1** based on a significant difference in task completion time for the Tetris task but not the NanoXML task. However, without improvements to the tool, this effect may be limited to experts.

5.2 The Tool did not Help perform Harder Tasks

We expected the NanoXML task to be more difficult for participants, and the task completion rate supports this expectations; comparing completion rates between the Tetris and NanoXML tasks for groups A and B, nearly twice as many participants completed the Tetris task, as compared to NanoXML: 18 (Tetris) versus 10 (NanoXML).

For Hypothesis 2, we expected to see that participants would receive more benefits when using a tool on a *harder* task. To test this hypothesis, we compared the ratio of completion times for each participant's Tetris and NanoXML tasks. If the hypothesis were true, we would expect participants in group B to complete the NanoXML task at a much faster rate (as normalized by their Tetris task time). The task performance ratios were as follows. Subjects in group A performed the Tetris task 2.5 times faster than the NanoXML task. Subjects in group B performed the Tetris task 1.3 times faster than the NanoXML task. These values are significantly different ($p < 0.02$) by a two-tailed t-test.

According to these results, statistical debugging with the tool was no more effective than traditional debugging for solving a harder task. Therefore, we found **no support for Hypothesis 2**. Overall, the results suggest that there might be several factors that can explain why the automated debugging tool did not help the NanoXML task. In the discussion ahead, we speculate what these factors may be.

5.3 Changes in Rank Have no Significant Effects

For Hypothesis 3, we wanted to explore the effect of rank on the effectiveness of the tool. To assess this hypothesis, we measured the effect of artificially decreasing and increasing the rank of the faulty statements. If this hypothesis were true, we would expect the effectiveness to *decrease* when dropping the rank and *increase* when raising the rank.

As discussed in Section 4.5, we tested this hypothesis by conducting an experiment with 10 new participants split into groups C and D. For group C, the rank of faulty statements was kept intact. For group D, the rank for the faulty statement in Tetris was lowered from position 7 to position 35. Similarly, the rank for the faulty statement in NanoXML was raised from position 83 to position 16. (The new ranks were selected in a random fashion.) This modification of the ranks should have improved the effectiveness of the tool for the NanoXML task, and hurt the effectiveness of the tool for the Tetris task, for group D in comparison to group C.

Comparing the average completion time of the Tetris task for groups C and D, we did observe that group D (12:36) was a little slower than group C (10:12). Surprisingly, for the NanoXML task group D was not any faster than group C despite the much lower rank of the faulty statement (16 versus 83). In fact, group D actually performed the NanoXML task slower than group C—15:12 for group C versus 18:30 for group D.

However, the differences in performance between the groups were not statistically significant. In fact, a comparison of the completion time ratio of Tetris to NanoXML yields the same exact average fraction (.79) for both groups. This suggests that both groups were very similar in performance. Lowering rank did not hurt the performance of group D on the Tetris task, nor did raising the rank for NanoXML help improve group D's performance.

Therefore, overall, the results provide **no support** for Hypothesis 3. This suggests that the rank of the faulty statement(s) may not be as important as other factors or strategies. The participants may be using the tool to find other statements that are near the fault, but ranked higher than the fault. Or they may be searching through the statements based on some intuition, thus canceling the effect of changing the relative position of the faulty statement. For example, four participants in group D, who had the rank of the faulty Tetris statement lowered, were able to overcome this handicap by visiting another statement in position 3 (previously 8) that was in the same function as the bug. This suggests that programmers may use the tool to identify starting points for their investigation, some of which may be near the fault. This would lessen the importance of correctly ranking the *exact* line of code with the fault.

5.4 Programmers Search Statements

To answer our Research Question 1 on patterns used by developers when inspecting statements, we analyzed the logs produced during the usage of the tool. Specifically, we wanted to assess whether developers inspected statements in order of ranking, one by one, or followed some other strategy. We used the navigation data collected from the 24 participants in group A and B, of which 22 had usable navigation data. We also examined the questionnaire of all 34 participants to gain insight into their strategies for using the tool. Based on this data, we have determined that **programmers do not visit each statement** in a linear fashion. There are several sources of support for this observation.

First, for each visit, we measured the delta between the positions of two statements visited in sequence. All participants exhibited some form of jumping between positions. Specifically, 37% of the visits jumped more than one position and, on average, each jump skipped 10 positions. The only exception were low performers (those who did not complete any task), whose majority (95%) cycled through the statements and very rarely skipped positions. Observing the number of positions skipped during all the visits, we hypothesize that smaller jumps may correspond to the skipping of blocks of statements; conversely, larger jumps seem to correspond to some form of searching or filtering of statements—a hypothesis also supported by the responses in the participants' questionnaires.

Second, the navigation pattern was not linear. Participants consistently changed directions (*i.e.,* they started descending the list, flipped around, and started ascending the list). We measured the number of "zigzags" in a participant's navigation pattern any time there was a change in direction. On average, each participant had 10.3 zigzags, with an overall range between 1 and 36 zigzags.

Finally, on our questionnaire given to all participants, many participants indicated that sometimes they would scan the ranked list to find a statement that might confirm their hypothesis about the cause of the failure, whereas other times they skipped statements that did not appear relevant.

5.5 No Perfect Bug Understanding

To investigate our Research Question 2 on the assumption of perfect bug understanding, we measured the tool usage patterns. We looked at the *first time* a participant clicked on the faulty statement in the tool, and then examined the participant's subsequent activity. If the faulty nature of a statement were apparent to the developers by just looking at it, tool usage should stop as soon as they get to that statement in the list.

We used the log data from the 24 participants in groups A and B and excluded data for participants that never clicked on the faulty statement, which left us with data for 10 participants. Only 1 participant out of 10 stopped using the tool after clicking on the fault. The remaining participants, on average, spent another ten minutes using the tool *after* they first examined the faulty statement. That is, participants spent (or wasted) on average 61% of their time continuing to inspect statements with the tool after they had already encountered the fault. This suggests that simply giving the statement was not enough for the participants to understand the problem and that **more context** was needed, which made us conclude that **perfect bug understanding is generally not a realistic assumption**.

5.6 Benefits and Challenges

To answer our Research Question 3, we analyzed the questionnaires completed by the participants. The analysis of the responses gave us a better understanding of our results, generated additional explanations for our observations, and identified potential benefits and challenges involved with using automated debugging tools. We analyzed this data for all 34 participants. Most participants reported that the primary benefit of the tool was to point them in the right direction. They would not necessarily use the tool to find the exact faulty statement, but rather identified several candidate causes for the failure. One participant, for instance, described the benefit of using the tool in the following way:

> "[The plugin] pointed some key statements, setting breakpoints on those helped find the problem."

The experiment results also highlighted a real difference in the tool effectiveness across tasks. Our analysis of the answer sheet identified several factors that can help explain this difference. For the Tetris task, participants could filter out unrelated statements more effectively based on their initial hypotheses. Because the failure description was related to rotation of a square figure, terms such as "rotation", "orientation", or "square" could be used to highlight relevant statements and ignore others, such as drawing routines.

For the NanoXML task, several participants indicated in their questionnaire that they would eventually abandon the tool after what they felt was too large a number of false positives. This suggests that there is a very rapid *interest drop-off* if developers cannot feel confident about the results they receive from the tool. For example, one participant described how he tried using the statement list, but quickly abandoned it after a few minutes:

> "The ranking list was too long and didn't help me with enough context. Actually, I know NanoXML and work with it, but [...] it was faster to use breakpoints."

Participants identified several challenges and possible enhancements to the functionality of the tool. We list a summary of those comments here, and discuss them in more detail in the following section. Several participants wanted the statements in the list to also include runtime values. Further, the participants wanted to integrate statements related to the exception chain with the statements in the ranking. Some participants wanted more program structure information with the statements (*e.g.,* method names). They also wanted alternative views, different ways of sorting and, in general, different ways of interacting with the data. Finally, many participants wanted some sort of explanation for the presence of a statement in the list and wanted to be able to trace a statement to its slices and related test cases.

6. DISCUSSION AND FINDINGS

In this section, we describe our findings about the behavior of programmers and discuss possible implications for future research in the area of debugging techniques and tools.

6.1 Programmers' Behavior

6.1.1 Programmers Without Tool Sometimes Fixed the Failure, Not the Fault

Ideally, when debugging, programmers are guided based on their understanding of the failure to a root cause. In practice, however, this does not always happen. Sometimes programmers discover that, through experimentation and manipulation of the program, they can stop the failure from occurring *without* actually fixing the fault.

We observed several occurrences of this phenomenon in our study. For example, in the Tetris task, the fault was due to an invalid configuration of the square figure's number of possible orientations. Quite a few participants did not correct this fault, but instead directly modified the rotation calculation code for the figures to bypass the failure. Interestingly, all these participants were in groups that used traditional debugging. No participants who used the ranking tool to fix the fault proposed this type of solution. We can therefore make the following preliminary observation.

Observation 1 - An automated debugging tool may help ensure developers correct faults instead of simply patching failures.

6.1.2 Programmers Want Values, Overviews, and Explanations

Developers used a mixture of the ranking tool (mainly for fault localization) and traditional debugging (mainly for fault understanding). If this observation is confirmed in further studies, this process could be more tightly integrated and streamlined. A common reason for engaging in traditional debugging with breakpoints, for example, was to acquire data values for statements. Since test cases have been already executed during automated debugging, including a way to display these values would cut this extra step and potentially speed up the use of the tool. As an added benefit, displaying values could give developers the ability to identify **suspicious values**.

For developers exploring unfamiliar code, an automated debugging tool can suggest many promising starting places. Even if the tool did not pinpoint the exact location of the fault, displaying relevant code entrance points could help program understanding considerably. This benefit strongly resonated with the developers in our study. One potential way to build on this strength would be to explore alternative interfaces for clustering and summarizing results. For an initial overview, for instance, developers may want to see suspiciousness aggregated by methods or files (a direction recently explored by Cheng and colleagues [2]).

We also identified a need to improve the explanatory capabilities of automated debugging tools. Developers were quick to disregard the tool if they felt they could not trust the results or understand how such results were computed. The ranking tool we used in the study provided developers with no traceability between ranked statements, relevant slices, and test cases. There was no way to identify which test cases were associated with a statement, or which statements belonged to the same dynamic slice. Better interactions with the underlying data could provide and connect with one another these additional pieces of information. If developers were equipped with such information, they would be empowered to explore the failure in a more methodic and data-driven manner.

In fact, experienced programmers use methodological and systematic methods of inspecting code [6, 24]. Unlike traditional debuggers or program slicers, most ranking based automated debugging techniques remove any source of coherence by mixing statements in a fashion that has no mental model to which the developer can relate. When using these tools, instead of working with the familiar and reliable step-by-step approach of a traditional debugger, developers are currently presented with a set of apparently disconnected statements and no additional support.

Observation 2 - Providing overviews that cluster results and explanations that include data values, test case information, and information about slices could make faults easier to identify and tools ultimately more effective.

6.2 Research Implications

6.2.1 Percentages Will Not Cut It

A standard evaluation metric for automated debugging techniques is to normalize the rank of faulty statements with respect to the size of the program. For example, assigning the faulty statement in NanoXML (4,408 LOC) with a rank of 83, when expressed as a percentage, suggests that only 1.8% of statements would need to be inspected. Although this result, at first glance, may appear quite positive, in practice we observed that developers were not able to translate this into a successful debugging activity.

Based on our data, we recommend that techniques focus on improving **absolute rank** rather than percentage rank, for two reasons. First, the collected data suggests that programmers will stop inspecting statements, and transition to traditional debugging, if they do not get promising results within the first few statements they inspect. For example, even when we changed the rank of the faulty statement in NanoXML from 83 to 16, there was no observed benefit. This is consistent with other research in search tasks, where it is clearly shown that most users do not inspect results beyond the first page and reformulate their search query instead [7]. Second, the use of percentages underscores how difficult the problem becomes when moving to larger programs. Percentages would suggest that we would not have to change our techniques, no matter whether we are dealing with a 400 LOC or a 4 million LOC program. From direct experience with scaling program analyses from toy programs to industrial-sized programs, we know that this is typically unlikely to be true.

Better measures can make sure we are using the appropriate metric for evaluating what, and to what extent, will help developers in practice. Otherwise, what may appear as a successful new debugging technique in the laboratory, could in reality be no more effective than traditional debugging approaches.

Implication 1 - Techniques should focus on improving absolute rank rather than percentage rank.

6.2.2 Focus More On Search

If current research is unable to achieve good values for absolute statement ranks, an alternative direction may be to enrich the debugging techniques by leveraging some of the successful strategies developers were observed to use. In particular, it may be promising to focus research efforts on how search of statements can be improved.

We observed that a common cause of frustration during debugging is the inability to distinguish irrelevant statements from relevant ones. According to reports from the participants, some developers successfully overcame this problem by filtering the results based on keywords in the statements. We found this to be key, as there may be some fundamental information in the developer's mind that, when combined with the automated debugging algorithm, may yield excellent results.

For example, in the NanoXML task, developers noted that using terms such as "index" or "colon" to filter through the results could help them find a result that matched their suspicion. In fact, had the developers searched for any of the terms "index", "colon", "prefix", or "substring", they could have filtered the statements so that the faulty one was within the first five ranked results. Unfortunately, performing this search manually among many results can be difficult in practice, whereas the combination of ranking and search could be a promising direction.

Besides combining search and ranking, future research could also investigate ways to automatically suggest or highlight terms that may be related to a failure. This would help in cases where a developer does not know the right terms to search for and could be done, for instance, based on the type of the exception raised or other contextual clues. It may even be that, given good search tools, developers could discover that the rank of a faulty statement does not matter as much as the **search rank**.

Implication 2 - Debugging tools may be more successful if they focused on searching through or automatically highlighting certain suspicious statements.

6.2.3 Grow the Ecosystem

The way performance (with respect to time) is computed in many studies makes assumptions that do not hold in practice. Like test suite prioritization, with automated fault localization the total time saved by configuring and using the tool should be less than the time spent using traditional debugging alone. In practice, at least in some scenarios, the time to collect coverage information, manually label the test cases as failing or passing, and run the calculations may exceed the actual time *saved* using the tool.

In general, for a tool to be effective, it should seamlessly integrate the different parts of the debugging technique considered and provide end-to-end support for it. Although some of these issues can be addressed with careful engineering, it may be useful to focus research efforts on ways to streamline and integrate activities such as coverage collection, test-case classification and rerunning, code inspection, and so on.

Implication 3 - Research should focus on providing an ecosystem that supports the entire tool chain for fault localization, including managing and orchestrating test cases.

6.3 Threats to Validity

We choose a time limit for tasks that made it possible to conduct our experiment within a two-hour time frame, so as to avoid exhausting participants. However, this time limit might have excluded less experienced participants who may need more time to complete the tasks. Our study has focused on more experienced developers, many of which could complete the tasks within the time limit, and may not generalize to novice users.

The participants of our study were students who may not have the same experience of professional developers, which could limit what can be concluded from the study. That said, some of the participants had several years of industrial experience as developers, and even more had at least some experience as developers during internships. Moreover, we observed that students with industrial experience and students without industrial experience performed comparably in the high-performers group.

A threat to external validity is the nature of programming tasks we selected and the fact that the developers were unfamiliar with the code being debugged. Programmers may not be able to effectively utilize their domain knowledge when debugging unfamiliar code. In practice, however, developers tend to work with both familiar and unfamiliar code, especially when they are part of larger teams. Although we could observe only one of these two cases, these types of tasks are representative of many real-world debugging activities (*e.g.,* bug fixes where developers need to understand someone else's code well enough to make a corrective change). Moreover, we believe that Tetris is a well understood and simple enough application that it can mimic the role of a familiar application for the developers.

Another threat to external validity is the fact that we considered only two subject programs and two faults within these programs. Our results may therefore not generalize to other programs and faults, and more experiments are needed to confirm our initial observations and analyses. Nevertheless, our results are promising and allowed us to make some interesting, albeit preliminary, observations that justify further studies and provide a methodology for conducting such studies.

A final threat to validity relates to the nature of the failure information. The fault for Tetris had no readily available entry point. In contrast, the fault for NanoXML had a stack trace available. Programmers without a tool, could use this stack trace as a starting point for their investigation. This difference might explain the success of the tool for Tetris and not for NanoXML. Several participants stated that, when debugging the Tetris failure, the tool helped pointing them in the right direction. Had a stack trace (*i.e.,* a starting point for the investigation) been available for Tetris too, the tool might not have been as beneficial. Nevertheless, our results show that when no such information is available, as it is common for failures that do not result in a crash or an exception, the tool can be helpful.

7. CONCLUSION

For 30 years, researchers have envisioned how automated debugging tools can help developers fix defects in code. In this paper, we had real programmers act this vision out and obtained both positive and negative results. In particular, developers could use a ranking based tool to complete a task significantly faster than without the tool, but this effect was limited to more experienced developers and simpler code.

One might argue that the point of ranking is not to help humans, but rather to measure and evaluate algorithms; only when algorithms are capable of consistently ranking faulty statements at the top should programmers start using the tools. But what if we never get to that point? Even when using an artificially-high rank, we found that the ranking tool considered was no more effective than traditional debugging for our more challenging task. If we are having trouble with small programs, what about large ones? The goal line for industrial-sized programs is moving faster and further away. The weaknesses observed in the ranking tool we used may indicate general shortcomings in today's automated debugging techniques that limit their effectiveness.

Based on our experience while working with 68 developers[1] that used our representative ranking tool, we have gathered several observations and identified several implications that may help define future research directions: empirical studies should use **absolute rank** for evaluating debugging algorithms, so as to account for interest drop-off in programmers' investigation and support scalability to larger programs; researchers should investigate the combination of search and ranking to associate a **search rank** to statements; tools should integrate the different activities involved in debugging and provide a **complete ecosystem** for debugging; finally, researchers should perform more human studies to understand how the use of **richer information** (*e.g.,* slices, test cases, values) can make debugging aids more useable.

Programmers have been waiting a long time for usable automated debugging tools, and we have already gone a long way from the early days of debugging. We believe that, to further advance the state of the art in this area, we must steer research towards more promising directions that take into account the way programmers actually debug in real scenarios.

Acknowledgments

This work was supported in part by NSF awards CCF-0964647, CCF-0916605, and CCF-0725202 to Georgia Tech.

8. REFERENCES

[1] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proceedings of the Symposium on Principles of Programming Languages (POPL 03)*, pages 97–105, New Orleans, LA, USA, 2003.

[2] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 09)*, pages 141–152, Chicago, IL, USA, 2009.

[3] H. Cleve and A. Zeller. Finding failure causes through automated testing. In *Proceedings of the International Workshop on Automated Debugging (AADEBUG 00)*, Munich, Germany, 2000.

[4] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 96)*, pages 121–134, San Diego, CA, USA, 1996.

[5] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10:405–435, 2005.

[6] M. A. Francel and S. Rugaber. The value of slicing while debugging. *Science of Computer Programming*, 40:151–169, 2001.

[1] This number includes participants in our pilot experiment.

[7] L. A. Granka, T. Joachims, and G. Gay. Eye-tracking analysis of user behavior in www search. In *Proceedings of the International Conference on Research and Development in Information Retrieval (SIGIR 04)*, pages 478–479, Sheffield, UK, 2004.

[8] A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain. In *Proceedings of the International Conference on Computer-Aided Verification (CAV 04)*, pages 453–456, Boston, MA, USA, 2004.

[9] T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. In *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 99)*, pages 303–321, London, UK, 1999.

[10] D. Hao, Y. Pan, L. Zhang, W. Zhao, H. Mei, and J. Sun. A similarity-aware approach to testing based fault localization. In *Proceedings of the International Conference on Automated Software Engineering (ASE 05)*, pages 291–294, Long Beach, CA, USA, 2005.

[11] J. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE 02)*, pages 467–477, Orlando, FL, USA, 2002.

[12] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the International Conference on Automated Software Engineering (ASE 05)*, pages 273–282, Long Beach, CA, USA, 2005.

[13] J. A. Jones, M. J. Harrold, and J. F. Bowring. Debugging in parallel. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 07)*, pages 16–26, London, UK, 2007.

[14] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the International Conference on Software Engineering (ICSE 08)*, pages 301–310, Leipzig, Germany, 2008.

[15] A. J. Ko and B. A. Myers. Finding causes of program output with the Java Whyline. In *Proceedings of the International Conference on Human Factors in Computing Systems (CHI 09)*, pages 1569–1578, Boston, MA, USA, 2009.

[16] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7:49–76, 2002.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2005)*, Chicago, IL, USA, 2005.

[18] C. Liu and J. Han. Failure proximity: A fault localization-based approach. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE 06)*, pages 286–295, Portland, Oregon, USA, 2006.

[19] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical model-based bug localization. In *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 05)*, pages 286–295, Lisbon, Portugal, 2005.

[20] C. Parnin and R. DeLine. Evaluating cues for resuming interrupted programming tasks. In *Proceedings of the International Conference on Human Factors in Computing Systems (CHI 10)*, pages 93–102, Atlanta, GA, USA, 2010.

[21] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the International Conference on Automated Software Engineering (ASE 03)*, pages 30–39, Montreal, Canada, 2003.

[22] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 97)*, pages 432–449, Zurich, Switzerland, 1997.

[23] K. D. Sherwood and G. C. Murphy. Reducing code navigation effort with differential code coverage. Technical Report TR-2008-14, University of British Columbia, 2008. `ftp://ftp.cs.ubc.ca/local/techreports/2008/TR-2008-14.pdf`.

[24] J. Sillito, K. D. Voider, B. Fisher, and G. Murphy. Managing software change tasks: an exploratory study. *Proceedings of the International Symposium on Empirical Software Engineering (ISESE 05)*, pages 23–32, Noosa Heads, Australia, 2005.

[25] I. Vessey. Expertise in debugging computer programs. *International Journal of Man-Machine Studies: A Process Analysis*, 23(5):459–494, 1985.

[26] M. Weiser. Program slicing. In *Proceedings of the International Conference on Software Engineering (ICSE 81)*, pages 439–449, San Diego, CA, USA, 1981.

[27] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[28] M. Weiser and J. Lyle. Experiments on slicing-based debugging aids. In *Proceedings of the Workshop on Empirical Studies of Programmers*, pages 187–197, Norwood, NJ, USA, 1986.

[29] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[30] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 06)*, pages 169–180, New York, NY, USA, 2006.

[31] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of the International Conference on Software Engineering (ICSE 03)*, pages 319–329, Washington, DC, USA, 2003.