Visualization of Test Information to Assist Fault Localization

James A. Jones, Mary Jean Harrold, John Stasko
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
{jjones,harrold,stasko}@cc.gatech.edu

ABSTRACT

One of the most expensive and time-consuming components of the debugging process is locating the errors or faults. To locate faults, developers must identify statements involved in failures and select suspicious statements that might contain faults. This paper presents a new technique that uses visualization to assist with these tasks. The technique uses color to visually map the participation of each program statement in the outcome of the execution of the program with a test suite, consisting of both passed and failed test cases. Based on this visual mapping, a user can inspect the statements in the program, identify statements involved in failures, and locate potentially faulty statements. The paper also describes a prototype tool that implements our technique along with a set of empirical studies that use the tool for evaluation of the technique. The empirical studies show that, for the subject we studied, the technique can be effective in helping a user locate faults in a program.

1. INTRODUCTION

Attempts to reduce the number of delivered faults[1] in software are estimated to consume 50% to 80% of the development and maintenance effort [4]. Among the tasks required to reduce the number of delivered faults, debugging is one of the most time-consuming [3, 15], and locating the errors is the most difficult component of this debugging task (e.g., [16]). Clearly, techniques that can reduce the time required to locate faults can have a significant impact on the cost and quality of software development and maintenance.

Pan and Spafford analyzed the debugging process and observed that developers consistently perform four tasks when attempting to locate the errors in a program: (1) identify statements involved in failures—those executed by failed test cases; (2) narrow the search by selecting suspicious statements that might contain faults; (3) hypothesize about suspicious faults; and (4) restore program variables to a specific state [10, page 2]. Our work addresses the second task—selecting suspicious statements that may contain the fault. To identify suspicious statements, programmers typically use debugging tools to manually trace the program, with a particular input, encounter a point of failure, and then backtrack to find related entities and potential causes.

There are a number of ways, however, that this approach can be improved. First, the manual process of identifying the locations of the faults can be very time consuming. A technique that can automate, or partially automate, the process can provide significant savings. Second, tools based on this approach lead developers to concentrate their attention locally instead of providing a global view of the software. An approach that provides a developer with a global view of the software, while still giving access to the local view, can provide more useful information. Third, the tools use results of only one execution of the program instead of using information provided by many executions of the program. A tool that provides information about many executions of the program can help the developer understand more complex relationships in the system. However, with large programs and large test suites, the huge amount of data produced by such an approach, if reported in a textual form, may be difficult to interpret.

In previous work [5], we presented a visualization technique that provides a global view of the results of executing a program with an entire test suite. The technique uses color to visually map the participation of each program statement in the outcome of the execution of the program with a test suite, consisting of both passed and failed test cases. Based on this visual mapping, a user can inspect the statements in the program, and identify potential faulty statements.

Agrawal, Horgan, London, and Wong presented a similar technique [1, 2] and an implementation, called χSlice. χSlice also presents a color mapping applied to the statements of the program under test. Agrawal et al.'s approach uses set operations to determine the representation of statements in the program. Our approach, while similar, provides a more complex mapping from testing results to the program representation which lets more information be conveyed in the source-code display. We discuss, in more detail, the similarities and differences between our approach and Agrawal et

[1]In our discussion, we use errors, bugs, and faults interchangeably.

`mid() {` `int x,y,z,m;`	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
`1: read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●
`2: m = z;`	●	●	●	●	●	●
`3: if (y<z)`	●	●	●	●	●	●
`4: if (x<y)`		●				
`5: m = y;`		●				
`6: else if (x<z)`	●				●	●
`7: m = y;`	●					●
`8: else`	●		●	●		
`9: if (x>y)`			●			
`10: m = y;`			●			
`11: else if (x>z)`						
`12: m = x;`						
`13: print("Middle number is:",m);`	●	●	●	●	●	●
`}` Pass/Fail Status	P	P	P	P	P	F

Test Cases

Figure 1: Example faulty program (left); test suite, code coverage, and test results (right).

al.'s approach in Section 4.

This paper presents the details of our visualization technique along with a description of a tool, Tarantula, that implements the technique. This paper also presents the results of two empirical studies that evaluate the technique to determine whether the visualization can help a user locate faults in the program. We performed the first study on a program with a number of faulty versions, each containing one known fault. We performed the second study on a program with a number of faulty versions, each containing from two to five known faults. These studies show that, for our subject program, versions, and test suite, our visual mapping does illuminate the faulty statements most of the time. The studies also show that the visual mapping rarely suggests suspicion of a fault in nonfaulty statements. These results suggest that the technique can help localize the faulty statements in a program.

The main contributions of the paper are

1. A presentation of a new technique for visually mapping the results of program testing that provides a global view of these results while still letting the user access local information. The visual mapping helps in locating faults in a program by illuminating possible faulty statements.

2. A description of a tool called Tarantula that implements the visualization technique. Tarantula gives the user a number of options for visualizing the testing results and the program itself.

3. The results of a set of empirical studies, performed using Tarantula. These studies evaluate the effectiveness of our technique for fault localization using a real C program and versions containing faults found in the field.

In the next section, we present our visualization methodology and illustrate it with an example. In Section 3, we present our empirical studies that illustrate the potential effectiveness of our technique. In Section 4, we present related research and compare it with our work. Finally, in Section 5, we conclude and discuss future work.

2. METHOD

Software testers gather large amounts of data about a software system under test. These data are used to identify failures, and to help locate faults in the system. The data are also used to demonstrate the exhaustiveness of the testing, and find areas of the source code not executed by the test suite, thus prompting the need for additional test cases. We have developed a technique that utilizes this data, commonly gathered by software testers and testing environments, to aid in the debugging process, specifically fault localization.

The fundamental idea of our approach is to create a visualization of the source code of a software system in which the individual statements are colored according to their participation in the testing. To facilitate our visualization, given a test suite T for a software system S and a test case t in T, our technique requires two types of information about the execution of S with t: pass/fail results and code coverage. Test case t *passes* if the actual output for an execution of S with t is the same as the expected output for t; otherwise, t *fails*. The *code coverage* for t consists of the source-code statements that are executed when S is run with t. Thus, the input to the visualization consists of three components: the source code for S; the pass/fail results for executing S with each t in T; and the code coverage of executing S with each t in T.

A sample input to the visualization is shown below.

```
1 P 1 2 3 12 13 14 15 ...
2 P 1 2 23 24 25 26 27 ...
3 F 1 2 3 4 5 123 124 125 ...
```

On each line, the first field is the test-case number, the second field is the pass/fail (P or F) information about the test case, and the trailing integers are the code coverage (statements executed) for that test case.

To best illustrate our technique, consider the erroneous program for determining the median of the three input values, as shown in Figure 1. In this program, the fault lies on line 7. There, the value m, representing the median, should be assigned the value of x rather than the value of y. Six

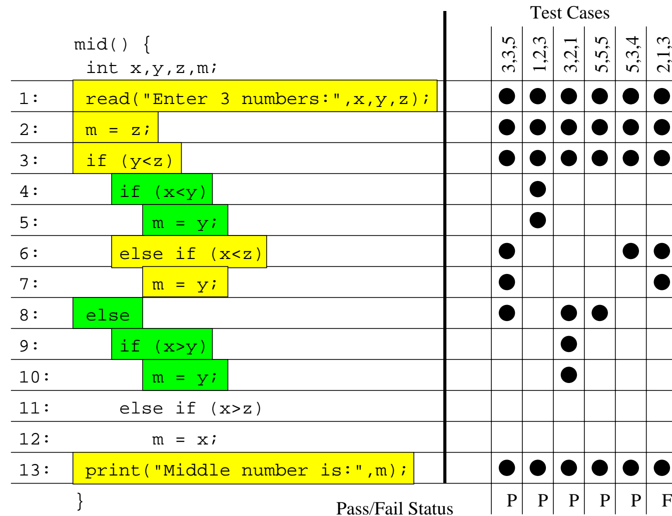

Figure 2: Example faulty program with source code colored according to the discrete three-color mapping.

test cases are supplied to the program as shown on the right in the figure: five of the executions produce the correct output and one of the executions produces an incorrect output. Each test case is represented by a column in Figure 1. The test-case input is at the top of each column; the test-case coverage of the statements in the program is shown by the black circles at the intersection of the column for the test case and the row for the statement; and the pass/fail result of the test case, listed as "P" or "F," respectively, is at the bottom of the column. For example, the first test case has an input of "3,3,5", executes statements 1, 2, 3, 6, 7, 8, and 13, and passes.

One approach to coloring the source code (which we initially tried) could use a simple color mapping: if a statement is executed only during failed executions, it is colored red; if a statement is executed only during passed executions, it is colored green; if a statement is executed during both passed and failed executions, it is colored yellow. We call this the *discrete* approach. Using this discrete color mapping, the source code for the program would be presented as in Figure 2. (Lines 1-3, 6-7, and 13 are colored yellow, and lines 4-5 and 8-10 are colored green; no lines are colored red in this example.)[2] Unfortunately, this simple approach is not very informative as most of the program is yellow, and the analyst is not given many helpful clues about the location of the fault.

Our approach uses a much richer visual mapping to assist the analyst in identifying the faults. The mapping consists of a continuous level of varying color (hue)[3] and brightness to indicate the ways in which the source-code statements participate in the passed and failed test cases. We call this the *continuous* approach.

2.1 Color Component

[2]Because color is central to our technique, we present this and other figures and charts throughout the rest of the paper in color. When the paper is viewed or printed in black and white, the figures and charts display in grayscale. The figures and charts can be seen in the color version of the paper at http://www.cc.gatech.edu/aristotle/Publications.

[3]Throughout the paper, we use "color" instead of hue.

The first component of the visual mapping is the color. Our technique utilizes color to indicate, for each statement in the program, the relative percentage of passed test cases that execute the statement to failed test cases that execute the statement. If a higher percentage of passed test cases executes a statement, the statement appears more green. Conversely, if a higher percentage of failed test cases executes a statement, the statement appears more red. Statements executed by nearly equal percentages of passed and failed test cases appear yellow.[4] The key idea is that the color of a statement can be anywhere in the continuous spectrum of colors from red to yellow to green. The intuition is that statements that are executed primarily by failed test cases should be highly suspicious as being faulty, and thus are colored red to denote "danger"; statements that are executed primarily by passed test cases are not likely to be faulty, and thus are colored green to denote "safety"; and statements that are executed by a mixture of passed and failed test cases do not lend themselves to suspicion or safety, and thus are colored yellow to denote "caution."

In particular, the color of a statement, s, is computed by the following equation:

$$\text{color(s)} = \text{low color (red)} + \frac{\%\text{passed(s)}}{\%\text{passed(s)} + \%\text{failed(s)}} * \text{color range} \quad (1)$$

In the equation, %passed(s) is a function that returns, as a percentage, the ratio of the number of passed test cases that executed s to the total number of passed test cases in the test suite. %failed(s), likewise, is a function that returns, as a percentage, the ratio of the number of failed test cases that executed s to the total number of failed test cases in the test suite. The value for the low end of the desired spectrum—in our case, red—is represented by "low color (red)." The "color range" denotes the value for the high end of the desired color spectrum—in our case, green—minus the value for the low color. For example in the image

[4]We conducted a series of iterative trials with a variety of different colors, and the red-yellow-green combination was found to be the most natural and the best for viewing.

	mid() { int x,y,z,m;	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
		Test Cases					
1:	`read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●
2:	`m = z;`	●	●	●	●	●	●
3:	`if (y<z)`	●	●	●	●	●	●
4:	`if (x<y)`		●				
5:	`m = y;`		●				
6:	`else if (x<z)`	●				●	●
7:	`m = y;`	●					●
8:	`else`	●		●	●		
9:	`if (x>y)`			●			
10:	`m = y;`			●			
11:	`else if (x>z)`						
12:	`m = x;`						
13:	`print("Middle number is:",m);`	●	●	●	●	●	●
	} Pass/Fail Status	P	P	P	P	P	F

Figure 3: Example faulty program with source code colored according to the continuous visual mapping.

manipulation and paint program, `gimp`, red is represented by a 0 value and green is represented by 120. In this case, the low color is thus 0 and the color range is 120 (high color value minus low color value).

If a statement is executed by 100% of the failed test cases and 50% of the passed test cases, for example, its color will be 1/3 of the way from pure red to pure green on a continuous color scale, thus making it an orange-ish color.

To more carefully represent the test data, the technique utilizes the percentage of test cases that execute a statement rather than the number of test cases that execute that statement. For example, if all 10 of the failed cases in a test suite execute a statement, but only 20 of the 200 passed cases execute that same statement, the statement will be a reddish-orange in our mapping (100% vs. 10%). However, the same statement would be a yellow-green if the technique simply considered the number of test cases (10 vs. 20).

2.2 Brightness Component

The second component of the visual mapping is brightness. Our techniques uses brightness to encode the percentage of coverage by either the passed or the failed test cases, whichever is higher. If all test cases in either set execute a particular statement (100%), then that statement is drawn at full brightness. If only a small percentage of the test cases executes a particular statement, then that statement is drawn very dark.[5] The intuition is that statements that are executed by a high percentage of the failed test cases are more likely to contain the fault than those that are executed by a small percentage of failed test cases, and thus are presented more brightly to attract the user's attention. Conversely, the statements that are executed by a high percentage of the passed test cases are more likely to be correct than those that are executed by a small percentage of passed test cases, and thus are presented more brightly to express the level of confidence in their correctness.

In particular, the brightness of a statement, s, is computed by the following equation:

$$\text{bright}(s) = \max(\%\ \text{passed}(s), \%\ \text{failed}(s)) \qquad (2)$$

Functions %passed(s) and %failed(s) are the same as those used in Equation 1. This mapping helps to encode the relative contribution of each statement during the testing. Consider two statements that are never executed in any passed test cases but are executed in some of the ten failed test cases. According to our color mapping equation, the statements must be a pure red because every test case that executes the statement fails. Suppose, however, that one statement is executed in all ten of the possible ten failed test cases and the other statement is executed in only one of the ten failed test cases. According to our brightness mapping equation, the first statement will be (maximally) bright red (100%), but the second will be a very dark red (10%), thus drawing the viewer's eye to the first statement.

Equation 2 uses the maximum of the two execution percentages rather than the combination of the two because we believe that it is a more faithful rendering of the execution data. If a statement was executed in four of the five failed test cases and in six of the 300 passed test cases in a test suite, our visualization will render the statement at 80% brightness, informing the viewer that a relatively high percentage of the failed test cases executed statement and it likely requires closer examination. If we combine the values, then the statement would be rendered at $10/305 = 3\%$ brightness, virtually black and unnoticeable.

Now, reconsider the simple program introduced in Figure 1. By applying the two rendering equations of our visualization technique to each statement, the presentation as shown in Figure 3 results. Lines 1-3 and 13 are colored yellow; lines 4-5 and 9-10 are colored dark green; line 8 is a brighter, but not the brightest, green; line 6 is orange-ish; and line 7 is orange-ish red.[2] This presentation highlights the potential (and actual) fault in the code more clearly than the one using the discrete approach as shown in Figure 2.

2.3 Prototype

Our continuous approach provides a presentation that can highlight suspicious statements in the program. Unfortu-

[5]In color space models, a pigment with zero brightness appears as black.

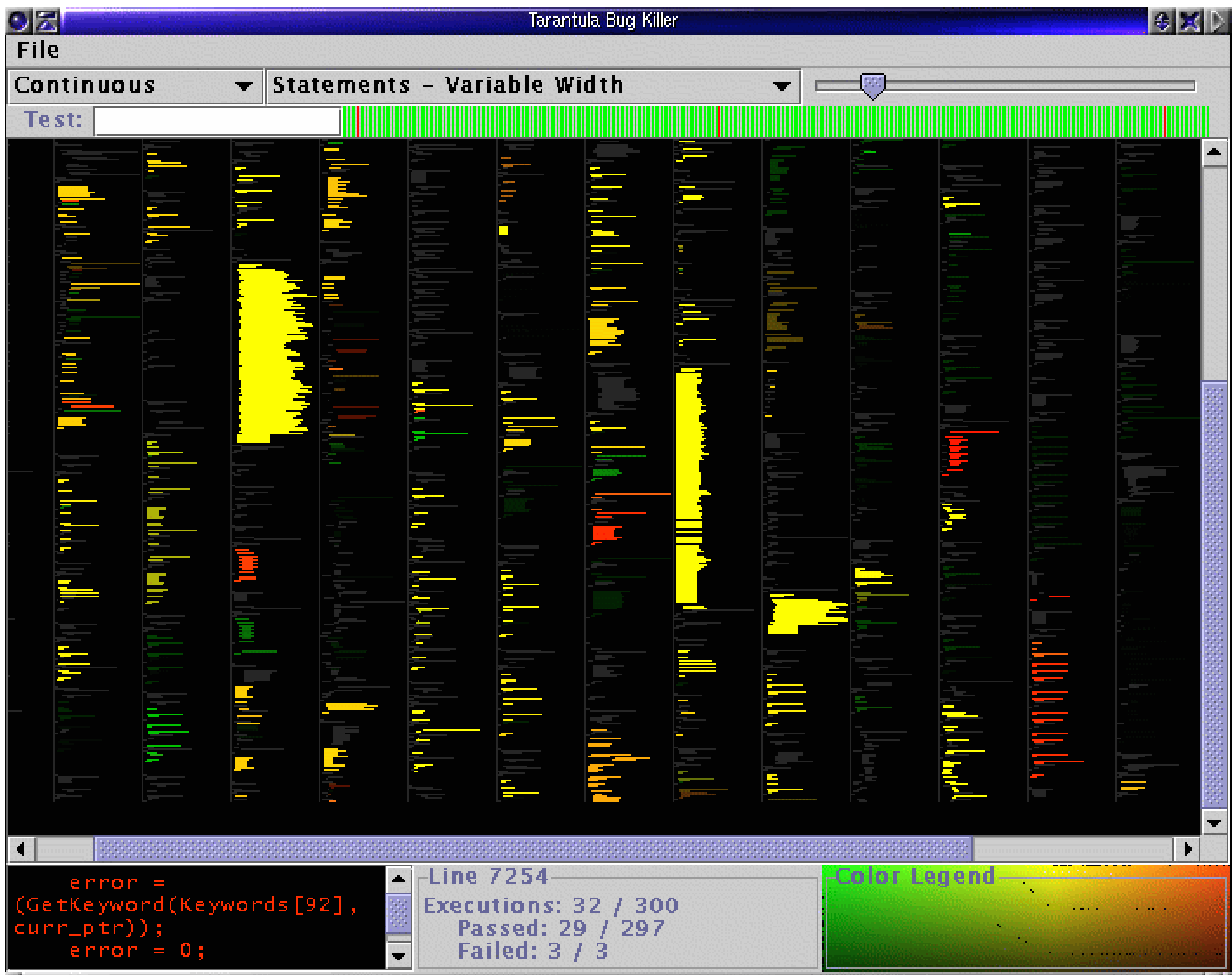

Figure 4: A screen snapshot of the TARANTULA system in Continuous mode.

nately, showing program source code as displayed in Figures 2 and 3 for any large system is problematic as it would require a tremendous amount of screen real estate. Thus, we utilize the visual mapping introduced by Eick et al. in the SeeSoft system [3, 6] to map each source-code statement to a short, horizontal line of pixels. This "zoomed away" perspective lets more of the software system be presented on one screen.

We have built a program visualization system, TARANTULA, that implements this visual mapping. TARANTULA is written in Java and consists of 3600 lines of code. It takes as input a software system's source code and the results of executing a test suite on the system. TARANTULA displays an interactive view of the system according to a variety of visual mappings. Figure 4 displays a screen snapshot of the system running.[2] The source code is displayed in the large center region.

The upper left has a pull-down menu (combo box) for selecting the display mode. Shown here in the Continuous mode. The system can display in six modes. The default mode simply shows the program source without coloration. The discrete mode uses the simple three-color mapping that we described above (the discrete approach). The continuous mode is the more complex mode utilizing the color and brightness equations described above (the continuous approach). Figure 4 shows TARANTULA in this continuous mode. There are three additional modes that let the user focus on only statements executed in passed, failed, or a combination of passed and failed (mixed) test cases. These three modes also use the continuous coloration and brightness.

The pull-down menu (combo box) to the right of the display-mode menu provides users with options for viewing the source code. There are currently three options. The first option displays the source code by statements and shows the statements using variable width (i.e., proportional to the length of the statements in the source code). In this view, the structure of the code is apparent. Figure 4 shows TARANTULA using this option. The second option displays the source code by statements and shows the statements using a fixed width (i.e., the length of all statements is the same). The third option displays the procedures in the

source code, colored with the color of the reddest statement in the procedure.

The slider in the upper right of the display controls the grayscale brightness of lines not being drawn using the red-yellow-green mapping (comments, unexecuted lines, filtered lines, etc.) In Figure 4, the slider is positioned to show those statements not involved in the mapping as light grey.

The text-entry box in the upper left, labeled "Test," provides a filter that lets the viewer enter the number or numbers of particular test cases. The display is then updated to show the source code as if those test cases were the only ones in the suite. Just to the right of the text-entry box is a long horizontal rectangle with thin vertical bands of red and green. Each band represents a different test case in the suite, in order, with red and green representing failed and passed test cases, respectively. Clicking on one of the bands selects that band as the focus, and the code coverage of that particular test case is shown in the main display area as colored lines of red or green.

When the mouse moves over a source-code line in the main display area, the source text for the line and a few surrounding statements are displayed in the lower left, and details of that statement's contribution in the entire test suite is shown in the lower center (pass and fail ratios). When the screen shot shown in Figure 4 was captured, the mouse was over line 7254 in the source code. For that line, 32 of the 300 test cases executed it: 29 of the 297 passed test cases and all of the failed test cases.

The color legend at the lower right includes a small black dot at each position in the color space occupied by a source-code statement. By rubber-banding a rectangle in this region around some points, the viewer can modify (filter) the main display area, showing and coloring only statements having the selected color and brightness values.

For more details on the system and the visual mappings, see [5].

3. EMPIRICAL STUDIES

To investigate the effectiveness of our visualization technique and guide our future work, we have designed a number of studies of Tarantula displays. The first set of studies investigates the effectiveness of our technique in illuminating the faulty statements in the program.

We used the `Space` program, which is an interpreter for an array definition language,[6] as the subject for our studies. `Space` is written in C and consists of 9564 lines of code (6218 of which are executable). `Space` has 33 associated versions, each containing a single fault that had been discovered during the program's development. We constructed a test pool for `Space` in two stages. We obtained an initial pool of 10,000 test cases from Vokolos and Frankl; they had created this pool for another study by randomly generating test cases [17]. Beginning with this initial pool, we instrumented the program for coverage and then supplied additional test cases to the pool until it contained at least 30 test cases that exercised nearly every statement and edge.[7] This process yielded a test pool of 13,585 test cases.

Using this pool of test cases, we extracted 1000 randomly sized, randomly generated, near-decision-adequate test suites from this test pool. This subject and these test suites have been used in similar studies (e.g., [7, 9, 12, 13]). These test suites are near decision-coverage-adequate: they covered 80.7% to 81.6% of the 539 conditions in the 489 decisions. The test suites ranged in size from 159 to 4712 test cases.

To investigate the effectiveness of our visualization techique in illuminating the faulty statements, we considered two questions:

1. How often does our technique color the faulty statement(s) in a program red or in a reddish color?
2. How often does our technique color nonfaulty statements in a program red or in a reddish color?

The first question helps us investigate how often our technique produces false negatives—that is, it fails to color the faulty statements red. The second question helps us investigate how often our technique produces false positives—that is, it colors nonfaulty statements red.

3.1 Study 1: Single-fault Versions

The goal of our first study is to determine the effectiveness of our technique in helping to localize the fault for programs that have a single fault.

We used twenty faulty versions of `Space`, each containing a different, single fault[8]; these were the twenty versions in which the fault resulted in a failure across all 1000 test suites. We executed each version with instrumentation on each of the 1000 test suites, and then applied the continuous color mapping technique to each version-test suite pair. The instrumentation gathered the coverage information about the test suites.

To answer the first question and report how frequently our technique colors the faulty statements red, we analyzed the color mapping applied to our subject. Figure 5 shows the results of this part of the study as a segmented bar chart. The chart contains one bar for each of the twenty versions of the program that we studied. Each bar in the segmented bar chart represents 100% of the faulty statements of that version of the program across all test suites. Each segment of each bar represents the number of times that the faulty statements were colored within a partition of the color spectrum. To simplify the presentation, we divided the color spectrum from red to green into ten equally-sized partitions, and thus there are ten segments in each bar.[2] We used the median color to represent each of these partitions; these colors and their partitions are illustrated and numbered in Figure 7.

The size of each segment represents the percentage of the times that the faulty statements fell into that color partition across all test suites. For example in Figure 5, across all 1000 test suites, version 20 (the rightmost bar in the chart) had approximately 3% of its faulty statements given colors within partition 1 of the red-to-green spectrum, 60% in partition 2, 34% in partition 3, 3% in partition 4, and less than 1% for all the rest of the color partitions (5-10).

[6] Alberto Pasquini, Phyllis Frankl, and Filip Vokolos provided the Space program and many of its test cases

[7] Only infeasible or extremely difficult to execute branches (such as those controlling an out-of-memory error condition) were not executed.

[8] A single logical fault may consist of multiple statements, or the absence of a necessary statement. In the latter case, we considered the statement preceding the absent statement to be the fault for the purposes of determining whether the users' attention may be directed to the location of the fault.

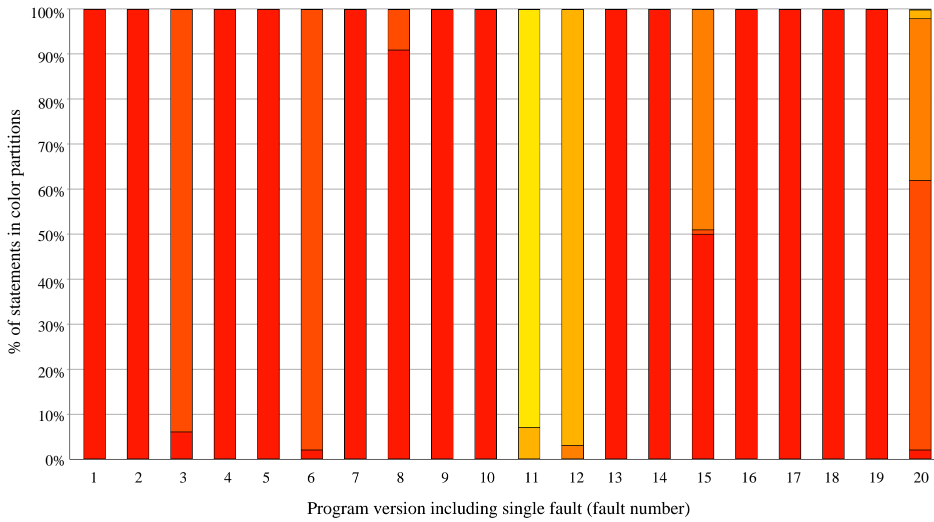

Figure 5: Resulting color of the faulty statements across all test suites for 20 versions of `Space`.

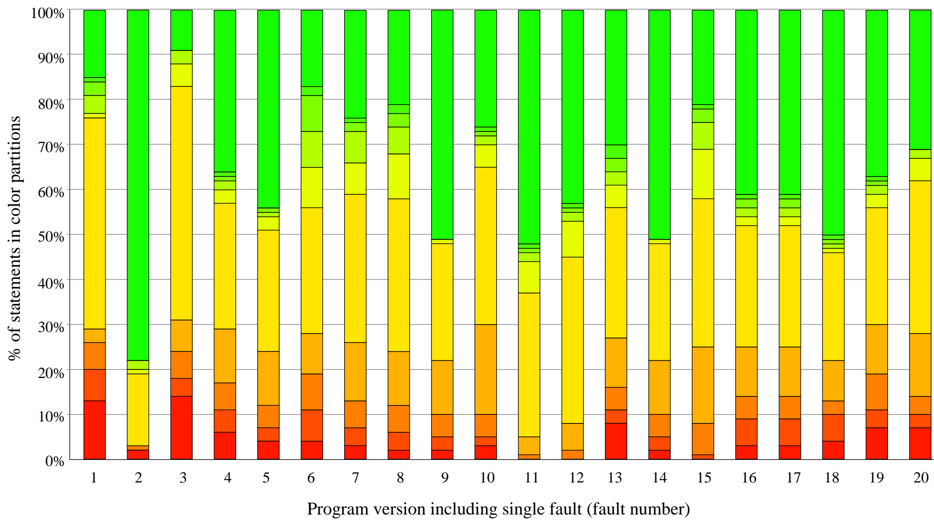

Figure 6: Resulting color of the nonfaulty statements across all test suites for 20 versions of `Space`.

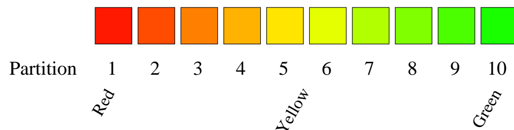

Figure 7: Legend for segmented bar charts showing color partitions of the red-to-green spectrum used for the segments of each bar chart.

Figure 5 shows that, for our programs and versions, most of the faulty statements across all 1000 test suites were colored with colors that are in the reddest three color partitions (1, 2, and 3) of the spectrum. However, for two versions—11 and 12—the faulty statements were colored with colors that are in the orange-yellow range of the spectrum (partitions 3, 4 and 5). We examined these two versions, and discovered that in them, the fault was in code that initializes variables in statements that are executed by all or most test cases. For these versions, the fault manifests itself as a failure later in the code.

To answer the second question and report how frequently our technique colors nonfaulty statements red, we again analyzed the color mapping applied to our subject. For this part of the study, we applied the same technique, and we display our results in the same fashion as in Figure 5 except that the segmented bar chart represents the nonfaulty statements, instead of the faulty ones.

Figure 6 shows these results. In all 20 versions, less than 20% of the nonfaulty statements are colored in the reddest 20% of the spectrum (partitions 1 and 2), and often much less, indicating that, for this subject, faults, and test cases, the technique significantly narrows the search space for the faults. Of the statements in the reddest 20%, we do not currently know how much is code immediately surrounding the fault. For example, if the statements immediately preceding and following the fault in the code listing are also colored red, the technique would still focus the user's attention to the faulty area of the code. In fact, we expect that a substantial number of these red statements are statements

immediately surrounding the fault in the code—those test cases that execute the fault likely (and in some cases, necessarily) also execute these surrounding statements. In future experiments, we will study the proximity of the nonfaulty red statements to the fault, using both source code locality and slicing techniques.

It is worth noting that versions 11 and 12, whose faults were colored in the yellow range (see Figure 5), have almost no red faulty or nonfaulty statements. This means that for these versions, our technique does not mislead the user, but simply fails to highlight the fault—no or few false positives. Perhaps these kinds of views—where all statements are yellow and green–are an indication that the fault lies in some main-line or initialization code. In the future we will attempt to address this situation by incorporating other types of analyses and new visualizations. Dependency analyses and slicing may provide useful information to overcome such situations.

3.2 Study 2: Multiple-fault Versions

The goal of our second study is to determine the effectiveness of our technique in helping to localize faults for programs that have multiple faults.

We again used the `Space` program but we modified it so that each of the faults from the single-fault study could be included or excluded with a compiler preprocessing macro. We used a script to create multi-fault versions of `Space` by randomly selecting the faults for inclusion or exclusion. For the experiment, we generated 40 versions of the program: 10 with two faults; 10 with three faults; 10 with four faults; and 10 with five faults. We executed each version with instrumentation on each of the 1000 test suites, and then applied the continuous color mapping technique to each version-test suite pair. The instrumentation provided the coverage information about the test suites. The results were recorded and analyzed in the same fashion as Study 1.

To answer the first question and report how frequently our technique colors the faulty statements red, we analyzed the color mapping applied to our subject. Figures 9(a)-9(d) show the results of this part of the study in the same segmented bar-chart manner as Study 1. As expected, the effectiveness of the technique declines on all faults as the number of faults increases. However, even up to five faults, the majority of the faults are colored in the red half of the spectrum (spectrum partitions 1 through 5). In fact, a large portion of the faulty statements are colored in the reddest three partitions (1-3). The decreasing effectiveness across all faults as the number of faults increases is a general trend, and is not always the actual behavior—the particular faults that are included are more important. This fact is exemplified by Figures 9(a) and 9(b). The 3-fault versions in (b) are generally more red than the 2-fault versions in (a). Had we used an additive approach in selecting faults—keeping the same faults of the $(n-1)$-fault versions and adding one new random fault for the n-fault versions—instead of randomly selecting all faults, we would expect this trend to be much more distinct. Overall, the decline in effectiveness in highlighting the faulty statements is less than we expected. Even up to five faults, the technique performed fairly well.

We, however, expected that the results of this study may be somewhat misleading. Because we are presenting the number of faulty statements in each partition for *all* faults in Figures 9(a)-9(d), colors for statements of individual faults are not distinguished. For example, the second bar of Figure 9(a) does not allow us to determine how the individual faults (14 and 17) fell into the different color partitions—did both fault 14 and fault 17 fall into both the green and red partitions, or, did one fault account for all of the green? We believe that this distinction is important because a fault that is not illuminated by the technique may eventually be illuminated if another more evident fault is located and removed. To investigate this situation, we plotted the data for each individual fault of a multi-fault version. From the left side of Figure 9, we chose a segmented bar that had both green and red segments (partitions 1 and 10) to dissect.

For this case study, we chose the 2-fault version containing faults 14 and 17—the second bar in Figure 9(a). Figure 8 shows the results of this case study. The first segmented bar in Figure 8 shows the colors assigned to the statement containing fault 14 across all 1000 test suites. The second segmented bar shows the colors assigned to the statement containing fault 17 across all 1000 test suites. Fault 14 is given a color in partition 1 (red) for 90% of the test suites, given a color in partition 2 for 1% of the test suites, and given a color in partion 10 (green) for 10% of the test suites. Fault 17 is given a color in partition 1 (red) for all 100% of the test suites. The final bar in Figure 8 shows the effect of rerunning TARANTULA on the program containing fault 14 after removing fault 17. Therefore, for this version, in the 10% of the test suites when only one fault is illuminated by the technique, the illuminated fault can be located and removed, thus allowing the technique to be reapplied to locate the second fault. This phenomenon further indicates the effectiveness of the technique, as at least one of the faults is illuminated for this version. In cases where fewer than all of the faults are revealed by this technique, the user could iteratively remove the discovered faults, retest, and reapply the technique until the test suite passes on all test cases.

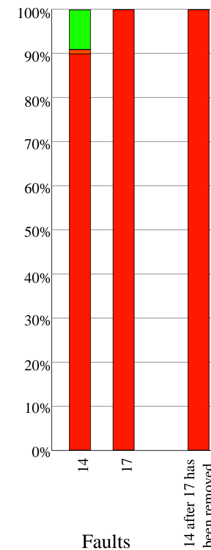

Figure 8: Resulting color for each individual fault in a 2-fault version (left); resulting color for the remaining fault after the discovered fault has been removed (right).

To answer the second question and report how frequently our technique colors nonfaulty statements red, we again analyzed the color mapping applied to our subject. Figure

9(e)-9(h) displays these results. For all multi-fault versions, we again notice the low number of red nonfaulty statements: less than 20% in color partitions 1 and 2, and often much less. This substantially reduces the search space of the program. We were surprised to find that there does not seem to be a trend toward more red nonfaulty statements as the number of faults increases in the program. It is still unclear how often the nonfaulty red statements are near the fault either in proximity or dependency. For example, the statements that immediately precede or follow the faulty statements may be red, in effect directing the user's attention to the faulty area of the program. Also, the code that calls a faulty procedure may be colored red. A user debugging the program may be directed to look at the code calling the faulty procedure and follow it to the fault. Based on our usage of TARANTULA, we have found that these are common cases. In future work, we plan to study the frequency of the closeness of the nonfaulty red statements to the fault, and develop techniques to illuminate this dependence.

Overall, our empirical studies indicate that the technique is effective in illuminating the fault or directing attention toward the fault, and narrowing the search space of the program for the fault. More studies need to be conducted on different subjects to study the generality of the approach, and to investigate further visualizations and analyses that can help the user in locating faults.

4. RELATED WORK

Eick et al. presented the SeeSoft system [3, 6] to display, with limited screen real estate, properties of large amounts of code. The "zoomed away" perspective provided by the SeeSoft technique gives a global view of the program that lets an abundance of information that has been mapped to the source code to be displayed. SeeSoft was used to display coverage information, but was not used to display the pass/fail results of the test cases on the program executed. We employ this SeeSoft technique to display the program under test. We provide a new color mapping—not provided by the SeeSoft system—to assist in fault localization.

Agrawal, Horgan, London, and Wong [1, 2] have also presented a technique that uses statement coverage (or execution slices) to aid in the task of fault localization. This technique is implemented in a tool called χSlice. Their technique also colors statements in a program to show their participation in passed and failed test cases. In Reference [2], Agrawal, et al., present a technique for locating faults that subtracts a single passed test case's execution slice from a single failed test case's execution slice. The resulting difference, or dice, is expected to contain the fault. In our usage of TARANTULA we have observed that this assumption is often incorrect. Figure 2 exemplifies one such case—if the first test case were chosen as the passed test case and the sixth test case were chosen as the failed test case, the difference would be the empty set.

In Reference [1], a brief description of a richer technique is described. This technique works by subtracting the union of the execution slices of the passed test cases from an execution slice of a single failed test case. The difference, or dice, is presented to the user in a pure red color. χSlice does show each statement that was in the intersection of the passed execution slices and failed execution slices in a gradient color scheme based on the number of passed execution slices to which the statement belonged; these statements in the intersection are presented as varying shades of blue. χSlice, however, uses only one failed test case's execution slice for the computation. Thus, using this technique, a statement, a, that is executed by one passed test case and 10 failed test cases, and another statement, b, that is executed by one passed test case and 1000 failed test cases, would be colored equivalently if both were executed by the failed test case picked for the dice. Our technique, on the other hand, uses all test cases in the test suite, both passed and failed, to attempt to localize the fault. This more complex mapping is richer and presents more data to the user.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented the details of our visualization technique that can assist in fault localization. Based on the results of executing a test suite for a faulty program, the technique provides a visual mapping of the participation of each program statement in the testing. To provide the visual mapping, the program statements are colored using a continuous spectrum from red to yellow to green: the greater the percentage of failed test cases that execute a statement, the brighter and more red the statement should appear. The paper also presented an overview of a tool that we have developed that implements our technique, along with the results of two studies that evaluate the effectiveness of the visualization technique. These results show that our technique is promising for helping locate suspicious statements—statements that may be faulty—and suggest some directions for future work.

First, the studies show that, for our subject program and versions, in many cases faulty statements are displayed in a color and a brightness that illuminates the fault. As expected, the faulty statements were easier to identify in the versions that had single faults—the versions used in Study 1. We were surprised, however, at how often at least one of the faulty statements was easy to identify in versions with multiple faults—the versions used in Study 2. We are conducting additional studies to further evaluate our technique. One set of studies will consider additional combinations of faults, producing more multiple-fault versions of `Space` with two to five faults; these studies will let us further validate our initial results. Another set of studies will consider versions with more faults; these studies will let us see how the technique works with very buggy programs.

Second, an interesting phenomenon that we observed in Study 2 was the coloration of some faulty statements in the green range. We realized that this occurred in cases where a test case that fails when executing one faulty statement may pass when also executing another faulty statement that happens to mask the effects of the first. However, a usual debugging process, in which faults are identified one at a time, can be helped by the visualization. Even though some faulty statements may appear green in the first view, other faulty statements will often appear in the red range. After the user finds and fixes one of these faults, reruns the test cases, and uses our visualization to view the results, another faulty statement is likely to be illuminated. The second part of Study 2, illustrated by Figure 8, demonstrates this process. We will conduct a study with our subject program and versions to see how often this occurs. For this study, we will start with versions containing a number of faults, and repeatedly determine if the visualization illuminates any of them, remove one of the illuminated faults, and rerun the

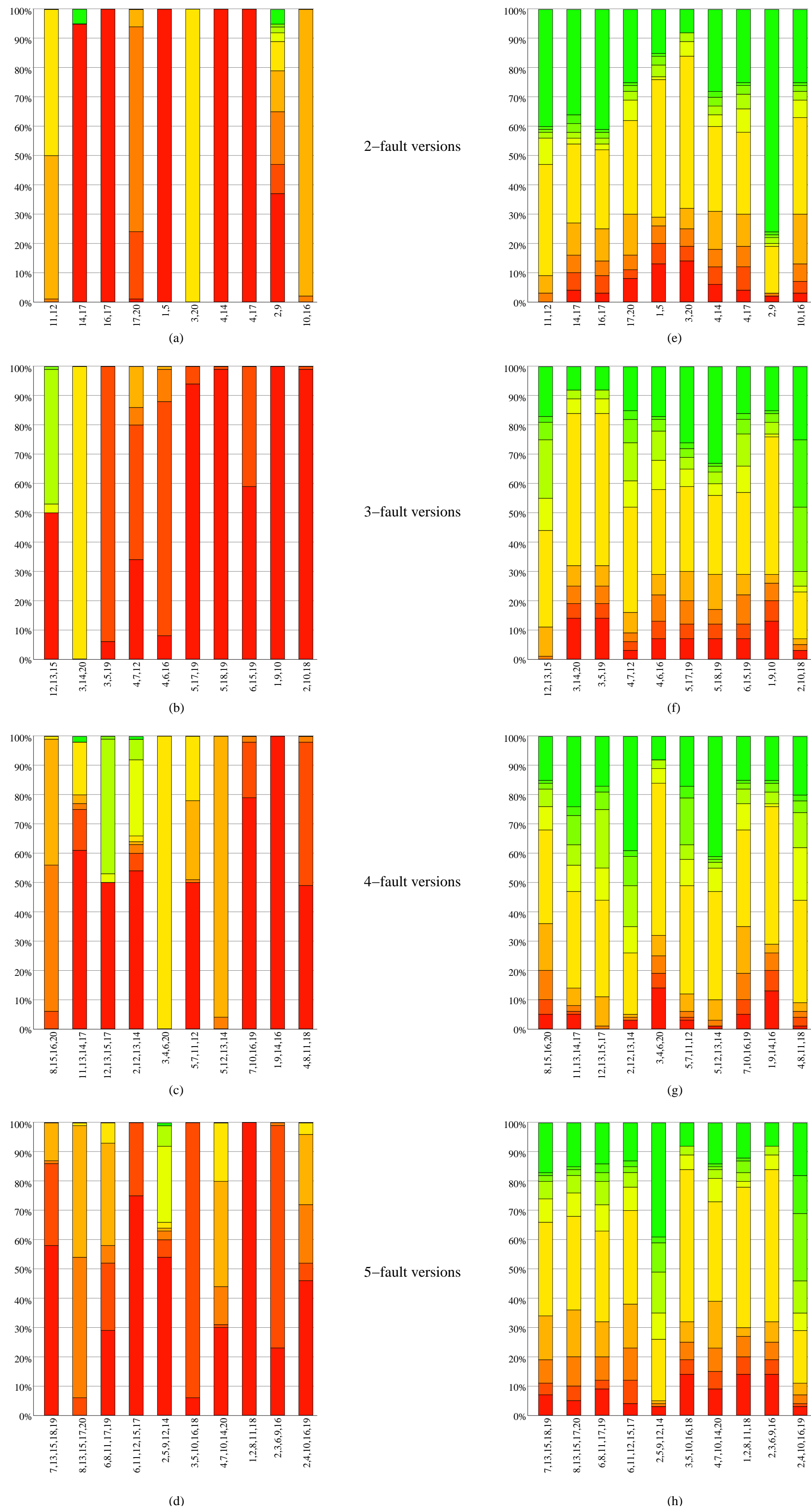

Figure 9: Resulting color for the faulty statements (left) and nonfaulty statements (right) in multiple fault versions across all tests suites.

test cases.

Third, both studies show that, although our visualization often illuminates the faulty statements (in the red range), sometimes there are no statements colored in the red range in the visualization. The bars in Figure 5, representing versions 11 and 12 for `Space`, illustrate this problem. We manually checked a few of these versions and found that they had faults in the initialization code or on the main path in the program. Perhaps the absence of a red area is an indicator that the faulty code is executed by most of the test cases in the suite. However, the number of data points precludes our drawing any conclusions about the meaning of this type of coloration. In future work, we will perform more experiments and determine why the faulty statements are not colored in the red range. We will also investigate other avenues to assist in fault localization when the current technique fails to highlight a fault.

Fourth, we were encouraged that, for our studies the technique significantly reduced the search space for the fault. Less than 20% of the nonfaulty statements were colored in the reddest 20% of the red-to-green spectrum, and often much less, while the fault usually resided in the reddest 20% of the spectrum. Both Study 1 and Study 2 illustrate this result. We want to investigate these red nonfaulty statements further and determine their "closeness" to the faulty statements. One approach we will consider is to use slicing to determine if they affect the faulty statement, and thus are always executed with it. Another approach is to determine if they are in the red range because they are in the same control-dependence region as the faulty statement. We plan to use these other analyses to provide additional information and views that can help locate faulty code.

Fifth, we plan to extend our study to include the brightness component of our visualization. In the studies presented herein, we focused on the color component of the mapping. The brightness component also needs to be studied to detect false positves and false negatives as was done for the color.

Sixth, we plan to improve the usability of TARANTULA in several ways. We will investigate approaches to providing a zoomable interface that lets the user view the results at various levels. This interface is necessary because we want to use our visualization for large software systems. Closely related to this zoomable interface is an approach to summarize useful information at different levels. Our current version of TARANTULA displays the source code at the statement and procedure levels. Clearly, more abstraction will be required for large systems.

Seventh, we plan to investigate the use of additional software visualizations [14] for use in fault localization. We also plan to explore the incorporation of further program-analysis information such as slicing, dependence, and control-flow data into TARANTULA in the future.

Finally, We plan to provide TARANTULA with the capability of editing and dynamic update. We will integrate an editor into TARANTULA so that a user can modify the source code after locating a fault. We also plan to modify TARANTULA so that it can rerun the modified program on the test suite, and dynamically update the view. To do this efficiently, we will incorporate our regression-test-selection technique [8, 11] into the system. This technique will select a subset of the test suite to run, based on the modifications.

6. ACKNOWLEDGMENTS

This work was supported in part by a grant from Boeing Aerospace Corporation to Georgia Tech, by National Science Foundation award CCR-9707792 to Ohio State University, and awards CCR-9988294, CCR-0096321, and EIA-0196145 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission. Alberto Pasquini provided the source code for the `Space`.

7. REFERENCES

[1] xSlice: A tool for program debugging. http://xsuds.argreenhouse.com/html-man/coverpage.html.

[2] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of IEEE Software Reliability Engineering*, pages 143–151, 1995.

[3] T. Ball and S. G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, Apr. 1996.

[4] J. S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software*, 9(3):191–195, 1989.

[5] J. Eagan, M. J. Harrold, J. Jones, and J. Stasko. Technical note: Visually encoding program test information to find faults in software. In *Proceedings of IEEE Information Visualization*, pages 33–36, October 2001.

[6] S. G. Eick, L. Steffen, Joseph, and E. E. Sumner Jr. Seesoft—A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, Nov. 1992.

[7] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the ACM International Symposium on Softw. Testing and Analysis*, pages 102–112, Aug. 2000.

[8] M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. S., S. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 312–326, October 2001.

[9] J. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *Proceedings of the International Conference on Software Maintenance*, pages 92–101, November 2001.

[10] H. Pan, R. A. DeMillo, and E. H. Spafford. Failure and fault analysis for software debugging. In *Proceedings of COMPSAC 97*, pages 515–521, Wahington, D.C., August 1997.

[11] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, Apr. 1997.

[12] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detecti on capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance*, Nov. 1998.

[13] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[14] J. Stasko, J. Domingue, M. Brown, and B. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

[15] Telcordia Technologies, Inc. *xATAC: A tool for improving testing effectiveness*. http://xsuds.argreenhouse.com/html-man/coverpage.html.

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

[17] F. Vokolos and P. Frankl. Empirical evaluation of the textual differencing regression testing tec hniques. In *International Conference on Software Maintenance*, November 1998.