Visualization for Fault Localization

James A. Jones, Mary Jean Harrold, and John T. Stasko
College of Computing,
Georgia Institute of Technology
Atlanta, GA 30332-0280
{jjones,harrold,stasko}@cc.gatech.edu

1 Introduction

Software errors significantly impact software productivity and quality. Attempts to reduce the number of delivered faults are estimated to consume between 50% and 80% of the development and maintenance effort [4]. Debugging is one of the most time-consuming, and thus expensive, tasks required to reduce the number of delivered faults in a program. Because software debugging is so expensive, researchers have investigated techniques and tools to assist developers with these tasks (e.g., [1, 3, 7]). However, these tools often do not scale to large programs or they require extensive manual intervention. This lack of effective tools hinders the development and maintenance process.

Studies show that locating the errors[1] is the most difficult and time-consuming component of the debugging process (e.g., [8]). Pan and Spafford observed that developers consistently perform four tasks when attempting to locate the errors in a program: (1) identify statements involved in failures; (2) select suspicious statements that might contain faults; (3) hypothesize about suspicious faults; and (4) restore program variables to a specific state [6]. A source-code debugger can help with the first task: a developer runs the program, one line at a time, with a test case that caused it to fail, and during this execution, the developer can inspect the results produced by the execution of each statement in the program. Information about incorrect results at a statement can help a developer locate the source of the problem. Stepping through large programs one statement at a time, however, and inspecting the results of the execution can be very time consuming. Thus, developers often try to localize the problem area by working backwards from the location of the failure (e.g., computing a slice). By considering all statements that affect the location in which an incorrect value occurred, a developer may be able to locate the cause of the failure. A source-code debugger can also help a developer with the fourth task: a developer can set breakpoints, reset the program state, and execute the program with the modified state. This process may help a developer concentrate on smaller regions of code that may be the cause of the failure.

Although these tools can help developers locate faults, there are several aspects of this fault-localization that can be improved. First, even with source-code debuggers and slicers, the manual process of identifying the location of the faults can be very time consuming. A technique that can automate, or partially automate, the process can provide significant savings. Second, because these tools lead developers to focus their attention locally instead of providing a global view of the software, interacting faults are difficult to detect. An approach that provides the developer with a global view of the software, while still giving access to the local view, can provide a developer with 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; such information is typically an artifact of the testing process. A tool that provides information about many executions of the program lets the developer understand more complex relationships in the system.

To address these problems, we have begun to develop visualization techniques and tools that can improve developers' ability to locate faults. Techniques and heuristics such as those described above may help a developer focus on areas of the program where faults may occur. However, with large programs and multiple faults, the huge amount of data produced by such an approach, if reported in a textual form, may be difficult

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

to interpret. A visualization can be more effective in summarizing results and highlighting promising locations in the program for further exploration.

As a first approach to this task, we have developed a visualization that provides both a high-level (global) view and a low-level (local) view of the system. The high-level view lets a developer focus on likely faulty sections of the code and gives the developer information about the results of a faulty program's execution on an entire test suite. The low-level view helps a developer locate the cause or causes of the faults.

2 Input Data

Developers and maintainers of large software systems usually create test cases for use in testing the systems. This testing provides evaluation of qualities such as correctness and performance. Each *test case* consists of a set of inputs to the software and a set of expected outputs from the execution of the software with those inputs. A set of test cases is called a *test suite*. It is not unusual for software engineers to develop large test suites consisting of unique test cases that number in the hundreds or even in the thousands.

Given a test suite T for a software system S and a test case t in T, we gather 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 lines that are executed when S is run with t.

The input to our 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 the execution of S on each t in T. Together, the second and third components can be viewed as an ordered list of the test cases. Each t in this list (1) is marked as "passed" or "failed," and (2) contains the code coverage for the execution of S with t. A sample input to our visualization system is shown below. On each line, the first field is the test case number, the second field is the pass/fail (P or F) information, and the trailing integers are the code coverage.

```
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 ...
```

Our challenge is to use this data to help software engineers find faults or at least identify suspicious regions in code where faults may lie. For large software systems with large test suites, this resulting data is huge, and is extremely tedious to examine in textual form. A visualization can summarize the data, letting software engineers quickly browse the test result representation to find likely problem regions of the code that may be contributing to failures.

3 Design Considerations

In developing our visualization tool, we had several key objectives. One was to provide a high-level, global overview of the source code upon which the results of the testing could be presented. We considered a number of alternatives and decided to use the "line of pixels"-style code view introduced by the SeeSoft system [5]. Each line of code in the program is represented by a horizontal line of pixels. The length of the line of pixels corresponds to the length of the line of code in characters, thus providing a far-away, birds-eye view of the code. Other objectives were to let viewers examine both individual test cases and entire test suites, to provide data about individual source-code lines, and to support flexible, interactive perspectives on the system's execution.

Our design's primary focus is on illustrating the involvement of each program line in the execution of the different test cases. We decided to use color to represent which and how many of the different test cases caused execution through each line. As we explored this idea further, the difficulty of selecting a good visual mapping became evident.

Suppose that a test suite contains 100 failed test cases and 100 passed test cases. Particular lines in the program might be executed by none of the test cases, only by failed test cases, only by passed test cases, or by some mixture of passed and failed test cases. Our first approach was to represent each type of line by a

different color (hue). Two different colors could represent passed and failed test cases, and a third color that is a combination of those two could represent mixed execution.

More flexibility was necessary, however. Consider two lines in the program that are executed only by failed test cases. Suppose that one line is executed by two test cases and the other is executed by 50 test cases. In some sense, the second line has more negative "weight" and could be represented with the same hue but with its code line darker, brighter, or more saturated than the first to indicate this attribute to the viewer.

This straightforward idea was sufficient for the pass-only or fail-only test cases, but was insufficient to represent lines executed by both passed and failed test cases. One approach was to vary the hue of the line, mixing combinations of the two extreme colors, to indicate how many test cases of each type executed the line. For example, suppose that a program line was executed by 10 failed and by 20 passed test cases. We could make its color closer to the color representing passed test cases since it was involved in twice as many of those test cases.

Unfortunately, this relatively simple scheme is not sufficient. Suppose that the entire test suite for the example above contains 15 failed and 200 passed test cases. Even though the line was executed by only half as many failed test cases (10 to 20), a much higher relative percentage of the failed test cases encountered the line ($10/15 = 67\%$ to $20/200 = 10\%$), perhaps indicating more "confidence" in that fact. Representing these ratios seemed to be more important than presenting the total quantities of test cases executing a line. Thus, the hue of a line should represent the relative ratios of failed and passed test cases encountered, and the color of this line would be more strongly the color indicating a failed test case.

This notion helped, but further issues arose. Consider two different source lines. The first is executed by 1 of 100 failed and 1 of 100 passed test cases. The two ratios are the same, thus the line's hue is a perfect blend of the two. Suppose that a second line is executed by 95 of 100 failed and 95 of 100 passed test cases. This line is the same hue, due to the equal ratios, but it seems desirable to render it differently because of its greater execution by the entire test suite. We needed to use a different attribute than hue to encode that fact.

Our visual interface makes concrete the heuristics hinted at above. We first experimented with a variety of background and line category colors by running a series of informal user tests. These studies helped us to select a color scheme using a black background with green representing passed test cases, red representing failed test cases, and yellow representing an even balance of passed and failed test cases. In the most advanced display mode, we decided to use hue to encode the ratio of the *percentage* (not quantity) of passed to failed test cases through a line, and to use brightness to represent the larger of the two percentages.

4 Tarantula

We have developed a system, called Tarantula, that depicts a program and its execution on a test suite. Tarantula's interface is shown in Figure 1. The middle area is the code-display area using the code-line representation pioneered in the SeeSoft system. The top area contains a number of interface controls for modifying the code-display area. The bottom area shows a detailed view of selected source, statistics of the selected source, and a color-space map.

The top area of the display contains a series of buttons, which are mutually exclusive controls for the display mode. The first mode, *Default*, simply shows the lines of code in dark gray in the code display area. The second mode, *Summary*, presents the most general, coarse summary of all the test information. Each line of the program is color-coded to reflect the outcome of the test cases that executed it. If no test case executed a line or the line is a comment, header, etc., the line is colored dark gray. If a line was executed only in successful test cases, the line is colored green. If a line was executed only in failed test cases, it is colored red. Finally, if a line was executed in both passed and failed test cases, then it is colored yellow.

The next three display modes (*Passes*, *Fails*, and *Mixed*) simply focus on one of the three types of cases in the summary mode and only color lines that meet those criteria. For example, in *Passes* mode, lines executed only in successful test cases are colored green and all others are colored gray. This effectively lets the viewer spotlight only those lines and focus more clearly on them. The *Fails* mode similarly shows only the red lines, that is, code lines only executed in failed test cases, and the *Mixed* mode shows only the yellow

Figure 1: An screen snapshot of the TARANTULA system in Shaded Summary mode.

lines executed in both passed and failed test cases. In each of these modes, the brightness for each line is set to the percentage of test cases that execute the respective statement for that mode.

The final mode, *Shaded Summary*, is the most informative and complex mapping. It renders all executed statements on a spectrum from red to green. The hue of a line is determined by the percentage of the number of failed test cases executing statement s to the total number of failed test cases in the test suite T and the percentage of the number passed test cases executing s to the number of passed test cases in T. These percentages are used to gauge the point in the hue spectrum from red to green for which to color s. The brightness is determined by the greater of the two percentages, assuming brightness is measured on a 0 to 100 scale. Specifically, the color of the line for a statement s that is executed by at least one test case is determined by the following equations.

$$\begin{aligned} \text{hue(s)} = &\quad \text{low hue (red)} + \frac{\%\text{passed(s)}}{\%\text{passed(s)}+\%\text{failed(s)}} * \text{hue range} \\ \text{bright(s)} = &\quad \max(\%\ \text{passed(s)}, \%\ \text{failed(s)}) \end{aligned}$$

For example, for a test suite of 100 test cases, a statement s that is executed by 15 of 20 failed test cases and 40 of 80 passed test cases, and a hue range of 0 (red) to 100 (green), the hue and brightness are 40 and 75, respectively.

The long, thin rectangular region located above and to the right of the code view area shows graphically the results of the entire test suite. A small rectangle is drawn for each test case from left-to-right and is color coded to its outcome—green for success and red for failure. This lets the viewer, at a glance, see the

overall trend within the test suite. Furthermore, the viewer can use the mouse to select any small rectangle in order to display only that test case's code coverage in the code view below. Also, the text-entry box in the upper left (labeled "Test:") lets the viewer enter the numbers of particular test cases and see the code coverage reflected in the code-display area.

The slider above the test suite display controls the brightness of the unexecuted statements. To make the unexecuted statements darker, the slider is moved to the left; to make them brighter, the slider is moved to the right. This feature lets the viewer gain familiarity with the code by making comments and other unexecuted code visible, and lets the viewer focus only on the executed code by making the unexecuted code black.

The bottom area of the display contains a color space map and detailed information about selected source code. The rectangle in the lower right is a map of the color space. Each statement is represented in this view as a black dot at the position for which its color has been determined in the current color mapping. The viewer is then able to see the distribution of all statements in the view, and can select statements by "rubber banding" them, thus causing the code view to be redisplayed for only those lines. For example, the viewer may wish to select all statements that are within 10% of pure red, or all statements that are executed by more than 90% of the test suite. Finally, clicking on a code line in the code display area makes it the focus: the source code near that line is shown in the bottom left of the interface, and the line number and test coverage statistics for that line are shown in the lower center.

5 Status

We are beginning to use Tarantula on large programs and preparing experiments to determine the effectiveness of our technique for locating faults in a program. For the cases when it is ineffective, we will need to include other program visualization views or supplement Tarantula's view with additional information to visually encode other program attributes such as control flow, call graphs, and dependence graphs. Along those lines, Ball and Eick created a visualization system that used the SeeSoft representation to encode program slicing information [2]. We plan to explore the addition of further program analysis information such as slicing, dependence, and control flow data into Tarantula in the future.

References

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

[2] T. Ball and S. G. Eick. Visualizing program slices. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pages 288–295, St. Louis, Oct. 1994.

[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.

[5] 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.

[6] H. Pan, R. A. DeMillo, and E. H. Spafford. Failure and fault analysis for software debugging. In *Proceedings of COMPSAC 97*, August 1997.

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

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