Visually Encoding Program Test Information to Find Faults in Software

James Eagan, Mary Jean Harrold, James A. Jones, and John Stasko

College of Computing / GVU Center
Georgia Institute of Technology
Atlanta, GA 30332-0280
{eaganj,harrold,jjones,stasko}@cc.gatech.edu

Technical Report GIT-GVU-01-09
June 2001

Abstract

Large test suites are frequently used to evaluate the correctness of software systems and to locate errors. Unfortunately, this process can generate a huge amount of data that is difficult to interpret manually. We have created a system called TARANTULA that visually encodes test data to help find program errors. The system uses a principled color mapping to represent how particular source lines act in passed and failed tests. It also provides a flexible user interface for examining different perspectives that show the effects on source regions of test suites ranging from individual tests, to important subsets such as the set of failed tests, to the entire test suite.

Keywords: Information visualization, software visualization, program visualization, debugging, testing, software development, fault localization

1 Introduction

Software errors significantly impact software productivity and quality. Attempts to reduce the number of delivered faults are estimated to consume 50% to 80% of the development and maintenance effort [CW89]. Debugging is one of the most time-consuming tasks required to reduce the number of delivered faults in a program. Thus, researchers have investigated techniques to assist with debugging (e.g., [BE96, Tel]). However, these techniques often do not scale to large programs or they require extensive manual intervention. This lack of effective techniques 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., [Ves85]). 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 [PDS97].

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 techniques can help programmers locate faults, there are several aspects of the process that can be improved. First, even with source-code debuggers, 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. 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 to interpret.

We are using information visualization [Spe01] and software visualization [SDBP98] techniques, together with data from program testing, to help software developers and maintainers localize faults in their code and subsequently remedy those bugs. Our techniques are more global in nature than previous approaches, providing a high-level overview of the software system and how it functions under testing, thus summarizing results and highlighting promising locations in the pro-

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

gram for further exploration. This paper presents a system we have developed, TARANTULA, that applies our techniques to depict a program along with the results of testing the program.

2 Input Data

Developers and maintainers of large software systems usually create tests (or test cases) for use in testing the systems. This testing provides evaluation of qualities such as correctness and performance. Each *test* 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 tests is called a *test suite*. It is not unusual for software engineers to develop large test suites consisting of unique tests that number in the hundreds or even in the thousands.

Given a test suite T for a software system S and a test t in T, we gather two types of information about the execution of S with t: pass/fail results and code coverage. Test 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 executing S with each t in T. Together, the second and third components can be viewed as an ordered list of information about the tests. 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 number, the second field is the pass/fail (P or F) information about that test, and the trailing integers are the code coverage for that test.

```
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 TARANTULA

3.1 Design Considerations

In developing TARANTULA, 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 [ESSJ92, BE96, Eic98]. 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 tests 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 tests. We decided to use color to represent which and how many of the different

tests 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 tests and 100 passed tests. Particular lines in the program might be executed by none of the tests, only by failed tests, only by passed tests, or by some mixture of passed and failed tests. Our first approach was to represent each type of line by a different color (hue). Two different colors could represent passed and failed tests, 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 tests. Suppose that one line is executed by two tests and the other is executed by 50 tests. 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 tests, but was insufficient to represent lines executed by both passed and failed tests. One approach was to vary the hue of the line, mixing combinations of the two extreme colors, to indicate how many tests of each type executed the line. For example, suppose that a program line was executed by 10 failed and by 20 passed tests. We could make its color closer to the color representing passed tests since it was involved in twice as many of those tests.

Unfortunately, this relatively simple scheme is not sufficient. Suppose that the entire test suite for the example above contains 15 failed and 200 passed tests. Even though the line was executed by only half as many failed tests (10 to 20), a much higher relative percentage of the failed tests 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 tests executing a line. Thus, the hue of a line should represent the relative ratios of failed and passed tests encountered, and the color of this line would be more strongly the color indicating a failed test.

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 tests. 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 tests. 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.

TARANTULA's 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 tests, red representing failed tests, and yellow representing an even balance of passed and failed tests. In the most advanced display mode, we decided to use hue to encode the ratio of the *percentage* (not quantity) of passed to failed tests through a line, and to use brightness to represent the larger of the two percentages.

3.2 System Capabilities

Figure 1 shows TARANTULA's interface acting on an example data set. 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 perspective in the program code display area. The bottom area shows a detailed textual view of a selected source-code region, statistics of the selected region, and a color-space map. One of the goals of TARANTULA is interactivity and flexibility, and different attributes of the data can be highlighted through different display modes and by mousing over or selecting different source-code lines. We next describe the systems's functionality

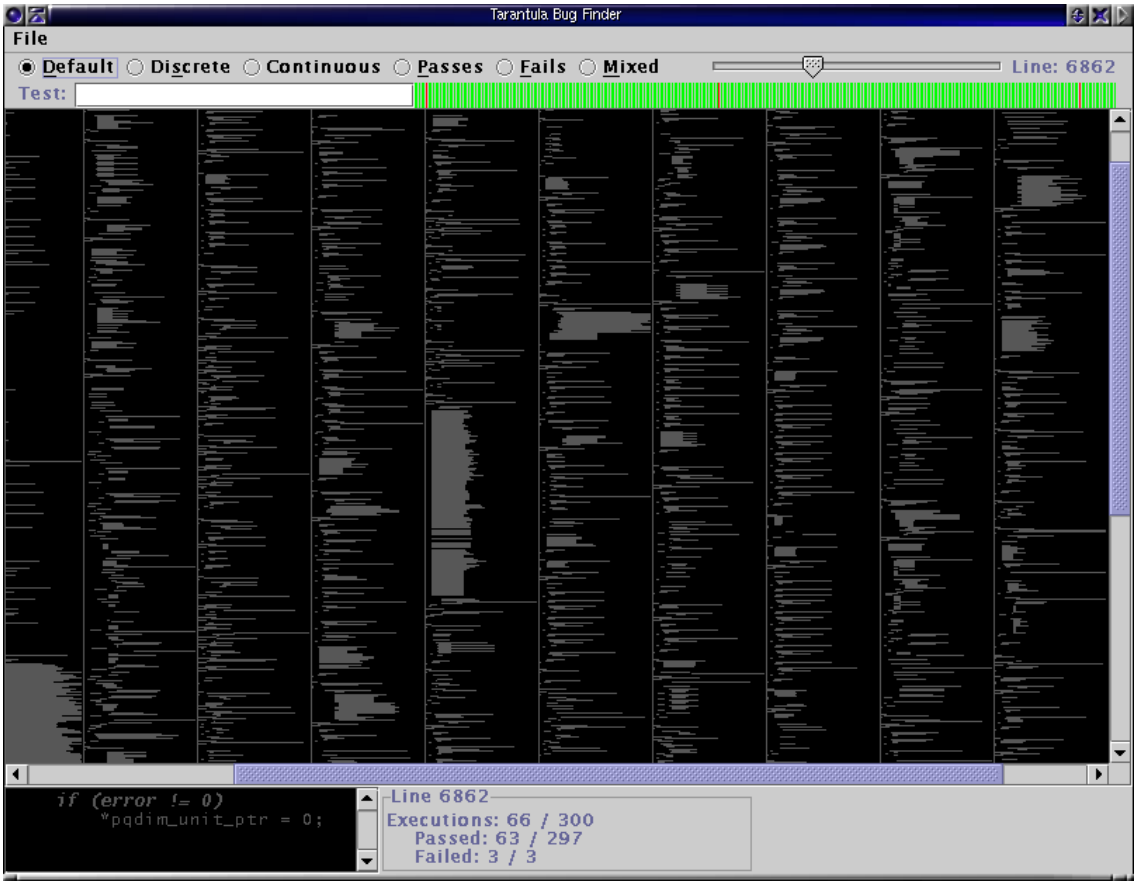

Figure 1: TARANTULA's "Default" view of an example system. Source lines are shown in gray.

in detail.

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 gray in the code display area and does not illustrate any testing data. This is the representation shown in Figure 1. The darkness or lightness of the gray is controlled through the upper-right slider, which is discussed later.

The second mode, *Discrete*, presents a general summary of all testing information in a straightforward manner. Figure 2 shows the example data in this mode. We call it "Discrete" because three discrete colors, red, yellow, and green, are used to color program statements to indicate how they were executed by the test suite. More specifically, each line of the program is simply color-coded to reflect the outcome of the tests that executed it. If no test executed a line or the line is a comment, header, etc., the line is gray. If a line was executed only in passed tests, the line is green. If a line was executed only in failed tests, it is red. Finally, if a line was executed in both passed and failed tests, then it is yellow.

The third mode, *Continuous*, is the most informative and complex mapping, and its view of the example data is shown in Figure 3. Unlike the Discrete mode, it renders all executed statements on a spectrum from red to green and with varying brightnesses. In particular, the hue of a line is determined by the percentage of the number of failed tests executing statement s to the total number of failed tests in the test suite T and the percentage of the number passed tests executing s to the number of passed tests in T. These percentages are used to gauge the point in the hue spectrum

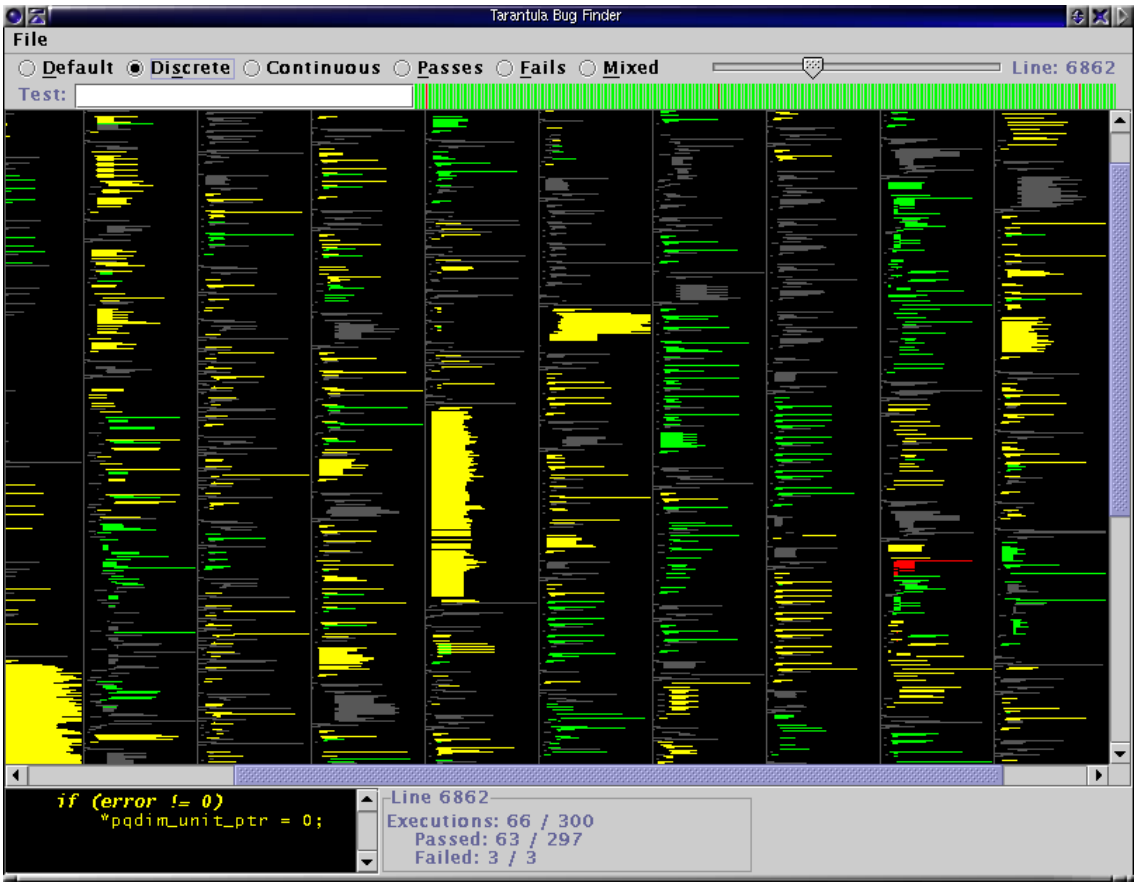

Figure 2: TARANTULA's "Discrete" view showing lines only executed in passed tests (green), only in failed tests (red), and in both passed and failed tests (yellow).

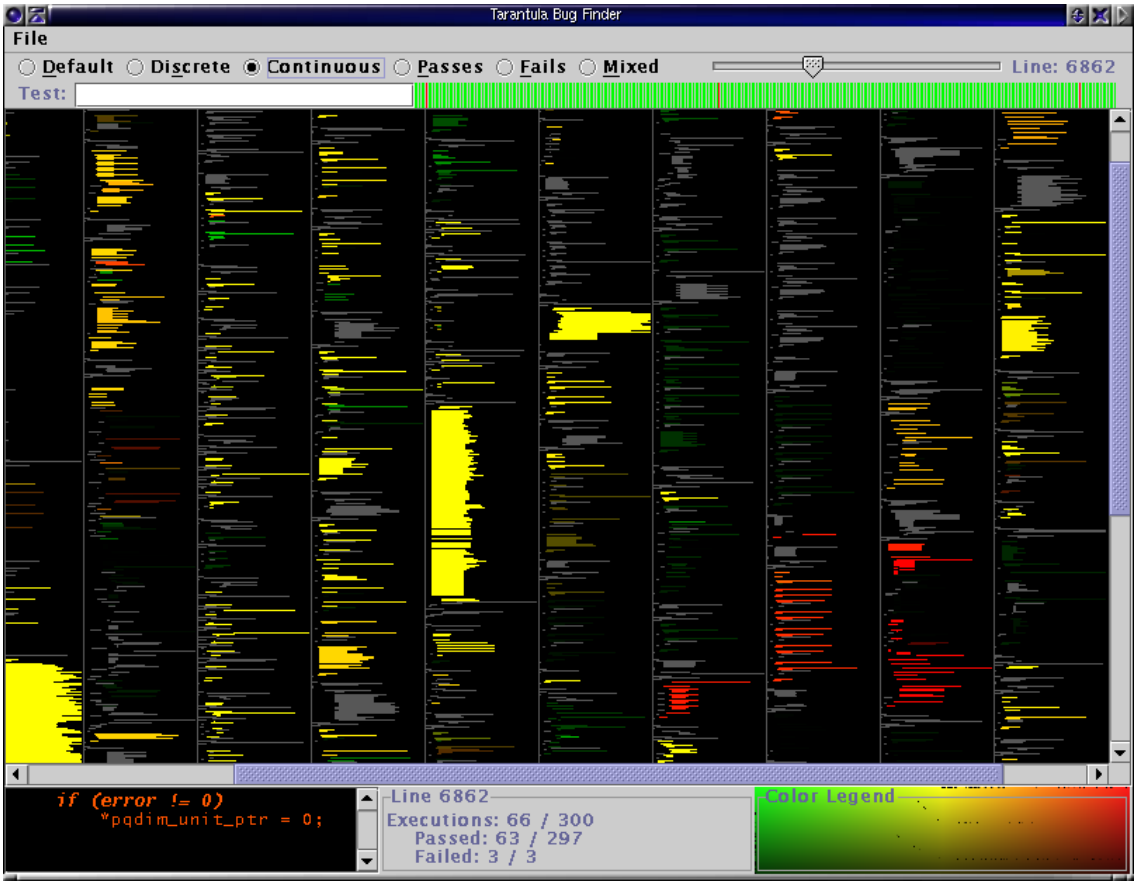

Figure 3: TARANTULA's "Continuous" view using both hue and brightness changes to encode more details of the test cases executions throughout the system.

from red to green with 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 is determined by the following equations.

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

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

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

The last three display modes (*Passes*, *Fails*, and *Mixed*) simply focus on showing all the lines in one of the three components of the Continuous mode. The same coloration and brightness mapping as in the Continuous case is used, but only lines that meet one of the three criteria are colored. For example, in *Fails* mode, lines executed only in failed tests are red and all others are gray. This representation for our example data is shown in Figure 4. This effectively lets the viewer spotlight only those lines and focus more clearly on them. In each of these modes, the brightness for each line is the percentage of tests that execute the respective statement of the tests for that mode. Lines executed by all failed tests are bright red, for example, and lines executed only in a small percentage of the failed tests are dark red.

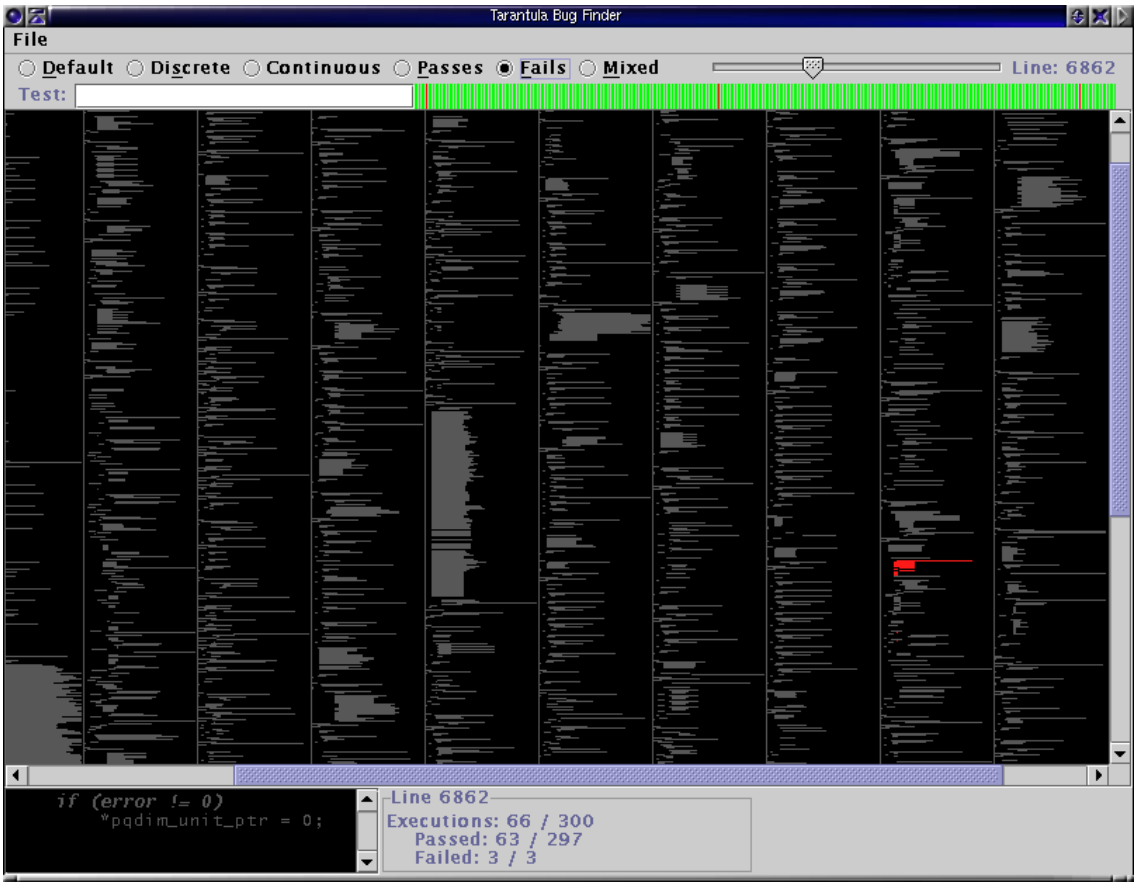

Figure 4: TARANTULA's "Fails" view showing lines executed only in failed tests and using the hue/brightness mapping of the "Continuous" view.

The long, thin rectangular region located above and to the right of the code-view area visually encodes the pass/fail attribute of each test in the suite. A small rectangle is drawn for each test from left-to-right and is color-coded to its outcome—green for pass and red for fail. This lets the viewer, at a glance, see the overall pass/fail trend within the test suite. Furthermore, the viewer can use the mouse to select any rectangle in order to display only that particular test'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 tests and see the code coverage of only those tests reflected in the code-display area.

As mentioned earlier, the slider above the test suite display controls the brightness of the unexecuted statements shown in gray. This feature lets the viewer gain familiarity with the code by making comments and other unexecuted code more visible (brighter gray), and then 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, when in Continuous mode, is a map of the color space. Statements are represented as black dots at the position corresponding to their color in the current color mapping. The viewer is then able to see the distribution throughout the color space of all statements in the view. The user also can select particular statements by "rubber banding" their dots in the map, thus forming a type of dynamic query that causes the code view to be redisplayed, coloring only appropriate 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, moving the cursor over 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.

To find faults in a system, a software engineer loads the input data about a system (described in Section 2) and can then examine the source code under a variety of perspectives. Presently, we are using TARANTULA to examine large programs under test to gain a better understanding of how program faults correlate to colored regions in the display. We need to determine whether faults usually fall in bright red regions of the display that indicate lines executed only in failed tests and in high percentages of those tests, or whether faults often lie in yellow regions executed both by passed and failed tests. Furthermore, we need to determine whether faults sometimes lie "upstream" or "downstream" of these colored regions. If so, we need to include other program visualization views or supplement TARANTULA's view with information to visually encode other program attributes such as control flow and calling relations. Along those lines, Ball and Eick created a visualization system that uses the SeeSoft representation to encode program slices [BE94, BE96]. We will explore the addition of program analysis information such as slices, into TARANTULA in the future.

4 Conclusion

This article presented an overview of TARANTULA, its user interface, and its visual encoding methodology for representing program test information. The research makes three main contributions. First, it introduces the idea of using a visual encoding of the potentially massive amount of program test result information to help software engineers locate faults in software systems. Second, it identifies a visual mapping using color and brightness to reflect each source line's influence on the test executions. Finally, it creates an informative and flexible user interface for presenting a variety of perspectives on the testing information.

References

[BE94] Thomas Ball and Stephen G. Eick. Visualizing program slices. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pages 288–295, St. Louis, October 1994.

[BE96] Thomas Ball and Stephen G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, April 1996.

[CW89] James S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software*, 9(3):191–195, 1989.

[Eic98] Stephen G. Eick. Maintenance of large systems. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, pages 315–328. MIT Press, Cambridge, MA, 1998.

[ESSJ92] Stephen G. Eick, L. Steffen, Joseph, and Eric E. Sumner Jr. Seesoft—A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[PDS97] Hsin Pan, Richard A. DeMillo, and Eugene H. Spafford. Failure and fault analysis for software debugging. In *Proceedings of COMPSAC 97*, pages 515–521, Wahington, D.C., August 1997.

[SDBP98] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

[Spe01] Robert Spence. *Information Visualization*. ACM Press, Pearson Education, Essex, England, 2001.

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

[Ves85] I. Vessey. Expertise in debugging computer programs. *International Journal of Man-Machine Studies: A process analysis*, 23(5):459–494, 1985.