Fault Localization using Visualization of Test Information

James A. Jones
Georgia Institute of Technology
jjones@cc.gatech.edu

1. Research Area

Fault localization using software visualization, testing information, and program analysis techniques.

2. Problem

Attempts to reduce the number of delivered faults in software are estimated to consume 50% to 80% of the development and maintenance effort [3]. Among the tasks required to reduce the number of delivered faults, debugging is one of the most time-consuming [2, 12], and locating the errors is the most difficult component of this debugging task (e.g., [13]). 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; (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 [8, page 2]. My research addresses the second task. 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.

This approach can be improved in a number of ways. 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. Thus, visualization techniques that effectively display large amounts of data can assist with interpretation.

3. Prior Research

Pan and Spafford [9] developed a set of heuristics that identify a set of suspicious statements in a program, but provides no ranking of the suspiciousness of the statements in the set. Thus, the user has no information about where to begin the debugging process within the set of statements. In addition, several of their heuristics require setting thresholds that must be tuned to the particular program and fault, which is not known until the faults are found. Also, their approach does not provide a way for the user to interface with the provided information for debugging.

Agrawal, Horgan, London, and Wong [1] present a technique that uses statement coverage (i.e., *execution slice*) to aid in fault localization. Their technique provides some visualization by coloring statements in a program to show their participation in passed and failed test cases. Their technique for locating faults subtracts a passed test case's execution slice from a failed test case's execution slice. The resulting difference (i.e., *dice*) is expected to contain the fault. My preliminary experiments have shown that this assumption is often incorrect—the faulty statements are often executed by some passed test cases. Thus, an effective approach requires some tolerance for cases in which the faulty statement is excuted by a passed test case.

Eick et al. presented the SeeSoft system [2, 4] to display 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 be displayed. Eick and colleagues used SeeSoft to display coverage information but did not use it to display the pass/fail results of test cases on the program executed.

4. Proposed Research

The goals of my work are to develop techniques that (1) partially automate the process of searching for faults, (2) provide a global view of the software as well as a local

one, and (3) use testing information from all or any subset of the test cases or executions. To do this, I will use a combination of testing and program analysis techniques coupled with information visualization. The approach will provide a visualization that determines a metric of "suspiciousness" and utilizing this metric, highlight those statements deemed suspicious. The approach will also provide a way to infer the number of faults in the program under test and help to identify the particular test cases that fail due to each fault. This information will let users focus on one fault at a time and help them determine the number and location of necessary breakpoints in a debugger. The approach will also provide a way for the user to explore the space by interacting with the data via dynamic queries. I will also develop a tool that implements these approaches. Using the tool, I will perform empirical evaluation to determine the effectiveness of the techniques and identify areas for future work. Based upon the results of the empirical evaluation, I will propose a methodology for debugging that utilizes my findings.

Work to date I have developed a technique and a visualization [5, 7] that displays the relative suspiciousness of statements in a program under test. My approach utilizes information about test cases that is already captured by many testing environments: the pass/fail status of each test case and the set of source code statements that it executes. The technique also inputs the source code for the program under test. Using this input, the approach creates a visualization of the program that presents a summary of how each statement was executed by the test suite while highlighting those statements that are deemed suspicious.

The first component of the visualization is the hue. My technique utilizes hue 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; 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. The key idea is that the hue of a statement can be anywhere in the continuous spectrum of hues 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 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 is represented by "low color (red)." The "color range" denotes the value for the high end of the desired color spectrum minus the value for the low color.

The second component of the visualization is brightness, which my technique uses 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, 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. 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.

The color for each statement is calculated and applied to a rendering of the software. I have defined three visual representations of the program at different levels of abstraction. The first and most detailed representation is the source code listing. Although this representation of the program is the most detailed, it does not let the user see much of the program at one time. Thus, I have defined representations of the program at higher levels of abstraction. The next level of abstraction is a view much like that of Eick and colleagues' SeeSoft view [2, 4]. In this representation, each source-code statement is mapped to a short, horizontal line of pixels. This "zoomed away" perspective lets more of the software system be presented on one screen while preserving the shape and structure of the source code. However, even for medium size programs, a significant amount of scrolling is still necessary to view the entire system. The most abstracted representation of the program utilizes the treemap view [11]. The treemap visualization is a two-dimensional, space-filling approach to visualizing a tree structure in which each node is a rectangle whose area is proportional to some attribute of that node. Treemaps are especially effective in letting users spot unusual patterns in the represented data. Moreover, treemaps can efficiently encode information for large-sized programs. In a 1280x1024 display, a treemap can visualize, on average, programs of more than 4,000 files [11]. Details of this unique mapping

to treemaps can be found in [7].

I conducted a study of this technique [5] and found that it performs well in most cases. The results also let me identify a number of areas for future work.

Future work One observation was that the technique seems to work better when there are fewer faults in the program. Thus, I plan to extend my work to find a subset of failing test case in the original test suite that fail due to a particular fault. One approach I will use is similar to Leon, Podgurski, and White's [6] clustering technique. Test cases that failed would be colored red and test cases that passed would be colored green. Those red executions that clustered together could be selected and the fault-localization technique could be applied to just those failing test cases as well as the passed test cases. This technique may narrow the set of test cases that failed due to a particular fault or a more restricted set of faults. Hopefully, this would improve the effectiveness of the localization technique. Studies would be performed to evaluate whether this technique improves the effectiveness of the localization technique in much the same way as the effectiveness was evaluated in [5].

Another observation was that the red (suspicious) statements are often in disparate parts of the program (i.e., different files and methods). Upon inspection, it is often apparent that the seemingly disparate parts of the program are actually related by control or data dependencies. For example, a method that is called from one red section of the program is also colored red due to the same test cases executing both. Observing the red statements in different parts of the code makes it difficult to determine the number of faults and the locations in which the user should place breakpoints in a debugger. One method that may be applied here is to provide a large-scale graph visualization (e.g., [10]) that shows whole-program control and data dependencies. In such a visualization, the deceptively disparate red sections of code may "collapse" together when they are truly related. This will let the user better determine the placement of breakpoints in a debugger and identify the number of faults in the program. Studies will be performed to investigate whether the related red sections of the code do in fact "collapse" together while the non-related sections of the code do not.

After I have thoroughly investigated and improved the technique for locating faults in a program, I would like to propose a debugging methodology that incorporates my findings. This methodology should decrease the total time that it takes to find and fix faults in a program. To validate this, I will have to perform studies on real users.

5. Contributions

My dissertation, when completed, will provide a number of contributions. First, it will provide a new approach for locating faults in programs that will be more effective and efficient than current techniques. This could provide savings that can significantly reduce the maintenance cost. Second, it will provide a toolset for locating faults and debugging software. This toolset will be useful for researchers that can use it for further experimentation, and practitioners that can use it for finding and fixing bugs. Third, it will provide a set of empirical studies that will evaluate the effectiveness of the approach and provide feedback for future work. Fourth, it will provide a new debugging methodology.

References

[1] 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.

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

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

[4] 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.

[5] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'01)*, pages 467–477, May 2001.

[6] D. Leon, A. Podgurski, and L. J. White. Multivariate visualization in observation-based testing. In *Proceedings of the 22th International Conference on Software Engineering (ICSE'00)*, pages 116–125, June 2000.

[7] A. Orso, J. A. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Proceedings of the ACM 2003 Symposium on Software Visualization*, pages 67–76, June 2003.

[8] 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.

[9] H. Pan and E. H. Spafford. Heuristics for automatic localization of software faults. Technical Report SERC-TR-116-P, Purdue University, July 1992.

[10] A. J. Quigley. *Large Scale Relational Information Visualization, Clustering, and Abstraction*. PhD thesis, University of Newcastle, Australia, August 2001.

[11] B. Shneiderman. Tree visualization with tree-maps: A 2-D space-filling approach. *ACM Transactions on Graphics*, 11(1):92–99, 1992.

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

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