The Challenge of Helping the Programmer During Debugging

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI. 02912
spr@cs.brown.edu

***Abstract*—Programmers spend considerable time debugging their systems. They add logging statements and use debuggers to run their systems in a controlled environment all in an attempt to understand what is happening as their program executes. Our hypothesis is that visualization tools can significantly improve the debugging process.**

A wide variety of tools have been developed for visualizing and understanding the dynamics of program execution. These tools can provide lots of information about executions. However, most tools are not designed to be used with a debugger. What is needed are tools that can work while the programmer is debugging a system and that provide the information the programmer needs to understand and assist the debugging process.

We have started to develop such tools within the context of the Code Bubbles development environment. However, there is much room for improvement and we call upon the software visualization community to think about and develop practical tools that will improve the debugging process.

Keywords*—*Software visualization, dynamic visualization, debugging.

I. Introduction

Debugging is one of the most difficult problems in software development. It is also one area in which software visualization can have the largest impact. The purpose of this paper is to challenge the software visualization community to develop practical tools that will actually be used by programmers to significantly improve debugging.

Interactive debugging and debuggers have not changed significantly over the last fifty years. They provide the programmer with the ability to set breakpoints, examine storage, and control execution through commands such as step, continue, and goto. The programmer then uses these commands to examine the execution in detail, attempting to determine where the program's behavior deviates from the expected behavior and what causes the deviation.

There are many reasons why debugging continues to be a difficult problem. Bugs often are exhibited in locations which are different both temporally and physically from where the incorrect code. The amount of relevant data that needs to be examined can be quite large and the differences that cause problems can be quite subtle. Some problems, such as data races, memory allocation issues, and iterations over hash tables, can be non-deterministic and vary between runs. Other problems, such as those involving performance can involve analyzing substantial amounts of data in the form of profiles and traces. Problems with graphics often involve complex interactions in unavailable library code.

Moreover, as programs become more complex, debugging is becoming more complex. Today's programs are typically multi-threaded, and often involve multiple communicating processes and multiple languages. Today's servers typically handle a range of transactions all at once, and the interaction of these transactions can cause problems. Programs that use graphic user interfaces are essentially non-deterministic since the order of input events generated by the operating system can vary from run to run.

In each of these cases, the difficulty in debugging can be viewed as a data problem. The programmer can obtain a large amount of data about the execution of their program, but then has to sift through this data to determine what is relevant and how the data relates to the program's expected and actual behaviors. Typically, programmers tend to minimize the amount of data they collect both to make this task feasible and practical. For example, programmers will set breakpoints and examine variables only at key points of the run, not on every statement; they will examine only those data structures they deem immediately relevant, not all data structures; and they will assume that parts of the program are behaving as expected until they see otherwise. Because of this minimization, complex debugging is generally an incremental process, with the programmer using information from one run to determine where to set breakpoints, what data to examine, and what assumptions to question in subsequent runs.

Ideally, the system should collect all the data the programmer is going to need during the debugging run and then make that data available to the programmer when needed in a way that makes the problems and issues apparent. The programmer should rarely have to rerun the system in order get the necessary understanding of what is happening. Moreover, the programmer should be alerted to potential problems that they might not know are occurring as they happen.

This work is supported by the National Science Foundation grant CCF1130822. Additional support has come from Microsoft and Google.

Software Visualization involves using a variety of techniques to effectively display large amounts of data about software so that the data that is of the most interest is highlighted and readily available. For example, city-based views of software systems make such problems as bloat, very large classes, overused classes or methods, excessive changes, etc. stand out as part of the visualization. Similarly, memory visualizations try to make unusual memory patterns such as memory leaks stand out.

Since much of debugging involves finding unusual or unexpected values in large volumes of data and software visualization is designed to handle just such problems, the combination of the two seems a natural marriage.

II. Prior Work

Many researchers (and a few companies) have attempted to use software visualization to assist in understanding the dynamics of program execution as summarized in [3,11], generally as a standalone tool either working off-line using traces such as with Jinsight [8,9] or on-line using instrumentation as with ExplorViz [4], but rarely within a debugger.

Some of the tools that have been tried include:

• Visual data structure displays including displays that animate or show changes as they are made. These were done within a debugger context by SGI, and are currently available in the ddd debugger front end [14].

• Performance tools that show where the program is spending its time such as Eclipse's TPTP package [5].

• Memory visualizations that show leaks or illegal memory access such as shown in Purify [6] and Plumbr [13].

• Lock monitoring tools that show the locking interactions between threads such as in Jive [10], the Java Lock Monitor or Java Mission Control from Oracle.

• Stack visualization tools that show the call stack over time such as those available in ExplorViz.

• Execution visualization showing what is currently executing by highlighting code lines or higher level views and high-level visualizations showing what the program is doing, for example as shown by various algorithm animation systems starting with Balsa [2].

• Specialized visualizations for particular classes of applications such as MPI message passing [12] or out-of-core algorithms [1].

In each of these cases, researchers have developed practical, useful, understandable visualizations that can be helpful to the programmer. However, if one looks at the tools provided by modern development environments such as Eclipse, one sees no visualization tools at all.

Based on our own experiences, we hypothesize several reasons that these tools have not been adapted by or are not widely used in modern environments. First, they often are too slow. Using the tools often slows down the program to the point where it becomes unusable or where debugging becomes painful. For example, attempting to use Eclipse TPTP on an interactive program, makes the subsequent interaction painful. (The slowdown is often a factor of 2-10 or more.) Memory and data structure viewers often have similar behaviors.

A second reason is that most of the tools are standalone, designed to monitor or provide information on programs running outside of the debugger. Many of the tools are off-line, collecting traces or other data while the program runs, and then processing and displaying that data after the run is complete. Even those that are on-line do not work well when they are applied to a program being debugged. For example, they don't understand when the program is stopped at a breakpoint, and the debugger can have difficulty correlating instrumented code with the user's source.

A third reason is that the visualizations do not do what they need to, that is, they do not highlight or make obvious the unexpected or relevant behavior. The primary reason for this is that what is relevant varies with the problem being debugged, which the tool typically doesn't know. For example, data structures in large systems, can be very complex, and often the low level details, e.g. how a list or table is actually implemented, are irrelevant, but not always. There can be many locks in a program used many times, but which are problematic can only be detected after a problem occurs. A large number of calling sequences and interactions are possible, but only a few are going to be apropos to a particular problem. Tools that are designed to provide an overview of what is happening are often difficult to use when looking at a very specific problem.

A fourth reason is that the tools are too complex to learn and use in the throw-away environment that is debugging. Programmers don't like to invest time in tools with unknown or limited benefits when they are concentrating on finding a bug. Complex visualizations can require a considerable learning curve, especially to isolate the particular data of interest to the programmer.

The value of using and providing additional, appropriate information to the programmer can be seen in the WhyLine project [7]. This runs the program in a controlled environment to gather enough information to be able to help the programmer pinpoint a potential problem by asking questions as to why certain things happened. A tool that gathered similar information while debugging (rather than off-line), and that could present the information to the programmer in a meaningful way, could be of immense benefit. This is true even if the tool were restricted to particular subset of debugging problems, for example memory or locking.

III. Our Current Work

We have been working on developing new visualization tools to aid debugging within the framework of the Code Bubbles development environment. This can be seen in Figure 1. The tools make use of a common framework that

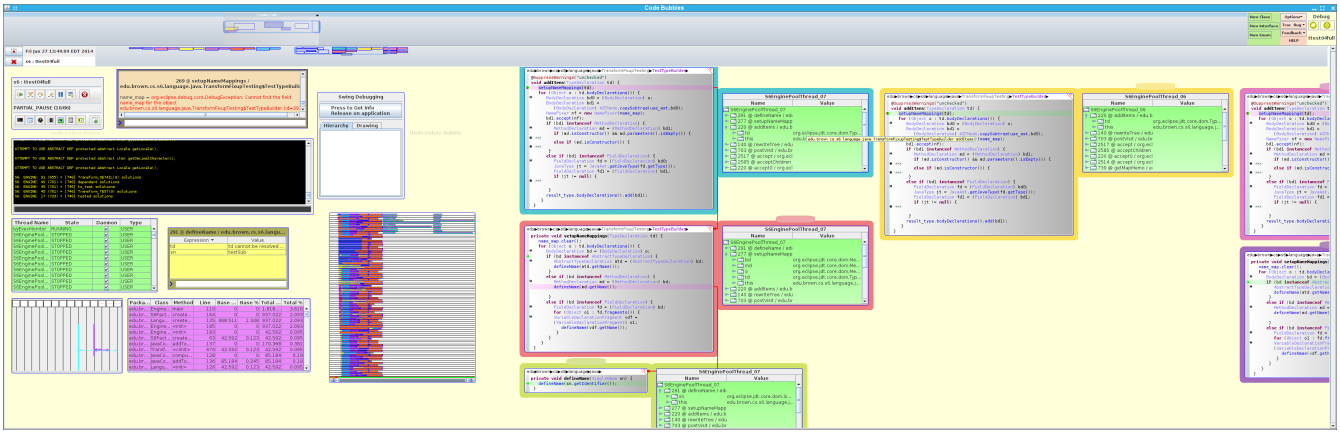

Figure 1. A view of debugging in code bubbles. The bubbles from the middle to the right show different stopping points in the program, with each pair of windows showing the source code for the function where the program along side a display of the relevant stack. Windows for a single thread are placed below one another, with a red line connecting them when one represents a call from the other. Separate stacks of windows represent separate threads.

uses minimal instrumentation along with stack sampling to gather the relevant information.

We started with the Code Bubbles interface for debugging. This creates a new bubble for new routines at a breakpoint, implicitly showing the call stack, with arrows between the bubbles showing the call relationship; it provides stack bubbles that can be frozen so that the programmer can examine variables from prior program states and compare them to current values; variables can be extracted from the stack into their own bubbles to show more detailed values; different threads are shown in different columns (or rows) of bubbles; a compact bubble shows the different threads and detailed state information.

In addition, we provide debugger bubbles (shown in more detail in Figure 2) to:

• Show the debugging history as a UML sequence graph. This is interactive in that the programmer can use it to view the code and variable values at any prior point in the debugging session.

• Show the execution history of the current thread when it has stopped at a breakpoint. This shows an approximation of what the thread was doing in the prior few seconds.

• Show information about a graphical user interface including the widget hierarchy and what routines are doing drawing at a chosen pixel.

• Detail where the program is spending its time executing through a table showing the time spent at various lines and methods.

• Detect and display detailed information about deadlocks when they occur.

• Display the value of programmer defined expressions and update at each breakpoint.

• Provide an interactive read-eval-print loop for the current context..

• Provide a high-level view of the history of execution in terms of threads, tasks and transactions that is generated automatically based on data collected during previous debugging runs.

All these tools are designed to work within the context of the debugger and to be continually active. The monitoring and data collection needed imposes an overhead of 1-2% (approximately 0.5 ms every 33 ms, but this varies by program and execution). Most of this is done in a separate thread in parallel with execution, and hence is almost invisible if extra cores are available on the machine. This allows the various tools to provide information about the past on demand, rather than requiring the programmer to proactively specify what information they want. Moreover, the tools that monitor execution, for example, the high-level viewer and the profiler, take breakpoints and user interactions with the debugger into account.

IV. The Challenge

Our work on Code Bubbles shows that it is possible to provide viable tools for debugging that involve significant amounts of information and that incorporate visualizations. However, the current set of tools is only a small first step and really doesn't address many of the problems programmers face when debugging. The particular problems with the current Code Bubbles debugging tools include:

• Many of the tools only gather detailed information at breakpoints, not continually. This limits the questions the programmer can ask, especially questions about the history of execution, and restricts the usefulness of the tools.

• The information on prior history, thread states, etc. is often too coarse to be of interest because it is gathered mainly by stack sampling.

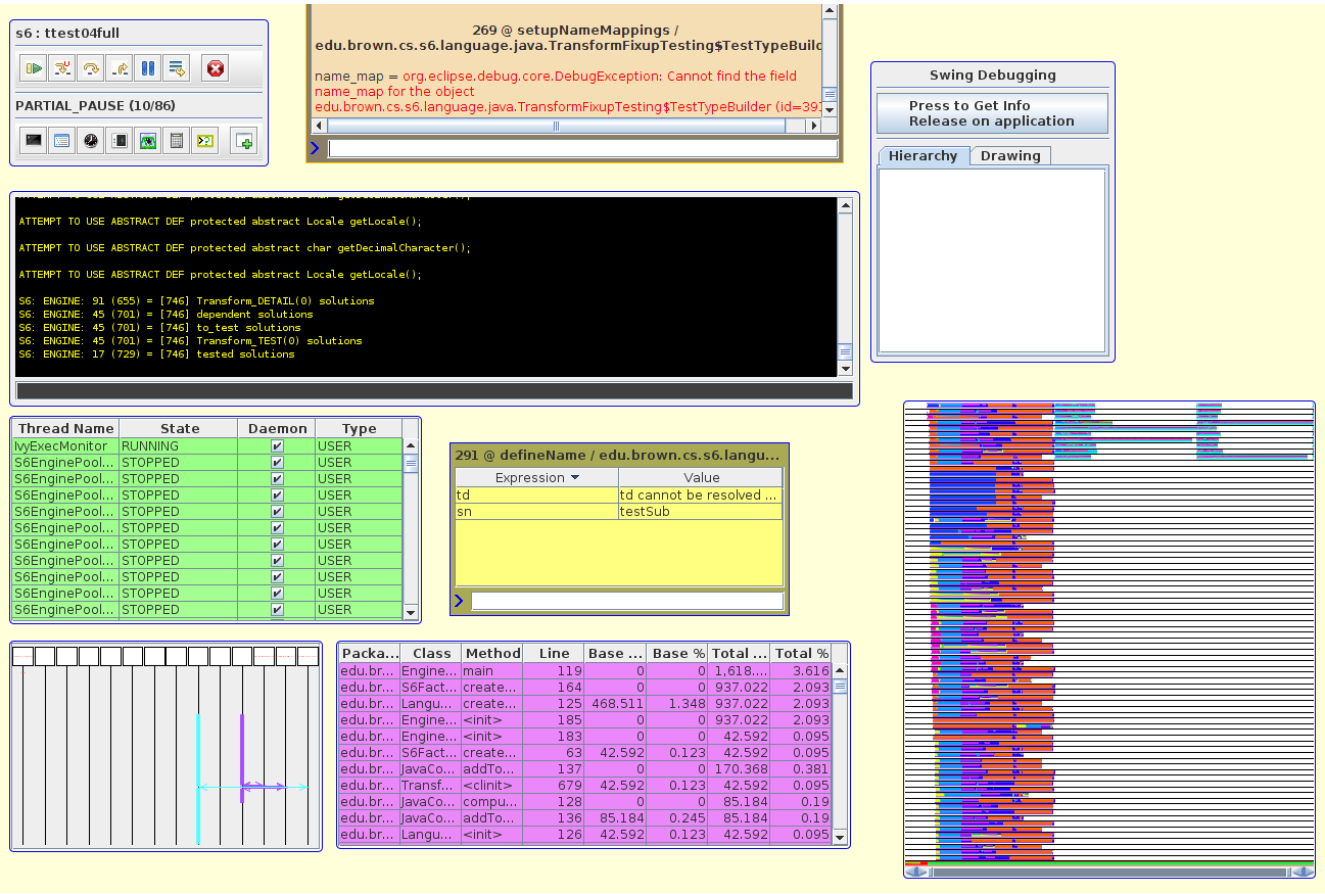

Figure 2. View of the various Code Bubbles debugging tools. The bubble at the top left is the debugger controller. To the right of this is a interaction window providing a read-eval-print loop for the current debugger context. Below these windows is the console for program input and output. Below the console on the left is a thread view showing all the current threads and their states; below on the right is a expression viewer where the user can enter expressions that are updated whenever the debugger gets control. The bubble at the bottom left shows the debugging session history as a UML interaction diagram, with different threads shown in different colors. The user can click on this to see the code and variable values at any point in the past. To the right of this is a performance view which is a sortable dynamic table showing where the program has spent its time during the debugging session. The view in the upper right is the Swing interaction that lets the user click on a pixel in Swing application to see the widget hierarchy and drawing commands issued for that particular pixel. Finally, the view at the bottom left shows the program execution for this particular debugging run in terms of threads, transactions and tasks. Each row in this figure represents a thread and the colored regions represent tasks and transactions being executed by that thread.

• Considerable information is available, but currently not displayed. For example, the debugger has stack and thread-based profiling data and the history of thread states over time, but does not display either. This information can be relevant to the programmer.

• Complex data structures, obvious candidates for software visualization, are displayed textually and are often difficult to delve into and view in detail.

• There are no tools to provide information about memory and locking behavior other than at deadlocks.

The end result is that while the tools available in Code Bubbles thus far are useful, they do not yet make a significant difference in the way debugging is done or in the complexity of the debugging process.

This leaves a prime opportunity for the software visualization community. To this end, we pose a set of questions to the community in the hopes of generating interest as well as new and better debugging tools. These are the basic questions that need to be answered in order to develop practical tools that will actually be used.

The first question is *what information could the debugger provide that would let programmers debug faster or more efficiently*? Can you speak from first-hand experience as to what would help you while debugging? Is this information about history, the current state or about tentative future executions? Is the information centered on control flow, dataflow, memory, locking, or other aspects of system behavior? Is the information very specific or is it general, and if it is specific, how might you specify what it is you are looking for? Many different types of information could be provided, but which are the ones that we should concentrate on.

The second question is, once we have determined the types of information that would be useful, *how to gather this information so that the debugger can address specific questions from the programmer in a timely fashion*. Suppose we can provide tools with a reasonable overhead budget, say 5% of execution time (which is still small enough to not be noticed). This can be combined with off-line analysis such as static analysis, information from prior debugging runs, and information from the programmer. While the exact information might be too large or too complex to gather, an approximation to the overall set that is still useful might be possible. Based on our previous experience looking at a wide variety of different software issues, and the advances in both hardware and software, I think this we will be able to answer this question in the affirmative.

The third question is determining *what visualizations can be provided that would offer meaningful data to the programmer while highlighting issues that the programmer should be aware of or that are relevant to the current debugging session*? This involves understanding how this data can be organized, analyzed, and presented in a meaningful way without overwhelming the programmer. It means specializing the displays to the particular problem being debugged, possibly with input from the programmer. It means developing displays that highlight the problems the programmer is looking for.

These three questions are intertwined. Understanding what visualization would be helpful to the programmer to improve their debugging experience tells us what information might be needed which then directs us to how to obtain relevant information. Understanding what information is needed or that can be retrieved can inspire new and better visualizations. The availability of new information can inspire both new visualizations of that information as well as ways of using this information to help the programmer.

The challenge then for the software visualization community is to provide a practical set of debugging tools that programmers will actually use in their everyday work and that can be used on real systems. We need to make using software visualization for debugging the standard practice of all programmers. This is a chance for software visualization to prove its value.

V. References

1. Tony Bernardin, Brian Budge, and Bernd Hamann, "Stack- based visualization of out-of-core algorithms," *Proceedings of SoftVis 2008*, (2008).
2. Marc H. Brown and Robert Sedgewick, "A system for algorithm animation," *Computer Graphics* **18**(3) pp. 177-186 (July 1984).
3. Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke, "A systematic survey of program comprehension through dynamic analysis.," *Technical Report TUD-SERG-2008-033, Delft University of Technology*, (2008).
4. Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring, "Live trace visualization for comprehending large software landscapes: the ExplorViz approach," *Proceedings of First IEEE Working Conference of Software Visualization*, (2013).
5. The Eclipse Foundation, "Eclipse test and performance tools platform project," *http://www.eclipse.org/tptp/index.php*, (August 2007).
6. Reed Hastings and Bob Joyce, "Purify: fast detection of memory leaks and access errors," *Proceedings Winter Usenix Conf*, (January 1992).
7. Andrew J. Ko and Brad A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behavior," *International Conference on Software Engineering 2008*, pp. 301-310 (May 2008).
8. Wim De Pauw, Doug Kimelman, and John Vlissides, "Visualizing object-oriented software execution," pp. 329-346 in *Software Visualization: Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, MIT Press (1998).
9. Wim De Pauw, Nick Mitchell, Martin Robillard, Gary Sevitsky, and Harini Srinivasan, "Drive-by analysis of running programs," *Proceedings International Conference on Software Engineering Workshop of Software Visualization*, (May 2001).
10. Steven P. Reiss, "JIVE: visualizing Java in action," *Proceedings International Conference on Software Engineering 2003*, pp. 820-821 (May 2003).
11. Steven P. Reiss, "Visual representations of executing programs," *Journal of Visual Languages and Computing* ***18***(2) pp. 126-148 (2007).
12. Lawrence Snyder, "Parallel programming and the Poker programming environment," *IEEE Computer*, pp. 27-36 (July 1984).
13. Vladimir Sor, Satish Narayana Srirama, and Nikita Salnikov-Tarnovski, "Memory leak detection in Plumbr," *Software: Practice and Experience*, (2014).
14. Andreas Zeller, "Debugging with DDD: user,s guide and reference manual.," *http://www.gnu.org/software/ddd/manual*, (2004).