The Paradox of Software Visualization

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912-1910
401-863-7641, FAX: 401-863-7657
spr@cs.brown.edu

Abstract

Software visualization seems like such a logical and helpful concept with obvious benefits and advantages. But after decades of research and work, it has yet to be successful in any mainstream development environment. What is the reason for this paradox? Will software visualization ever be actually widely used? In this paper we argue that most past and current work in the field (our own included) is out of touch with the reality of software development and that new approaches and new ideas are needed.

1. The Failure of Software Visualization

When we design and develop software we draw pictures. When we create data structures we draw pictures. When we explain software we draw pictures. When we teach software we draw pictures. When we debug software we draw diagram what is happening. When we analyze software we draw pictures. Pictures, diagrams, visualizations are central to the way we think about software.

Computers are good at drawing pictures. They can render graphs with hundreds of nodes and provide animated zooming and panning over graphs of thousands or more nodes. They can do high-density information visualization using statistical diagrams, dot plots, tree maps, and other technologies. They can create movies on the fly showing how things change.

Computer-based visualization and software should be an ideal match. Visualization should be central to our programming experience. Programmers should be using it when as they develop programs, when they analyze systems statically, when they look at the dynamic behavior of their systems, when they are debugging, when they are testing, and even when they are editing or modifying existing code.

Indeed there are examples where visualization is used in software development. Object-oriented design is generally done today using UML which consists of a variety of graph-based notations [3]. Developers who use UML draw use-case diagrams, class diagrams, state charts, interaction diagrams, etc. to describe their system. They also tend to enter a lot of text behind the diagrams using dialog boxes and the like to provide the necessary details. There have also been a number of successful domain-specific visual languages. One widely used example is LabVIEW from National Instruments for gathering and analyzing data from test and control instruments.

However these successful examples are relatively rare and are quite specialized. Programmers quickly give up the UML for header or interface files and ignore the UML notations once the program is written. Visual languages work well in specific domains but, despite much work, have had little success for general programming.

The areas where we expect visualization to have the most impact are in everyday software development and understanding. These are the areas that are the most expensive and important and where there is the biggest payoff. Yet this is were software visualization is not widely used.

None of today's mainstream programming environments feature visualization. None of the standard programming tools used today uses visualization in anything but the crudest way. The various experiments with visualization, even when they appeared in production environments, have failed; successor environments did not feel it was worth reproducing visualization tools. Programmers don't use visualization tools and don't seem to want to use them.

What is the reason for this paradox? Why have we as a community failed to achieve what seems to be a simple and obvious task? Is software visualization hopeless or just misguided? Where are we heading? Whither software visualization — or will software visualization wither?

These are the questions that we need to explore before we do more work in the field. These are the questions that should guide the future of software visualization.

2. Reasons for the Failure

We should begin by trying to understand why software visualization has not been successful. There can be any

number of reasons. It can be that our work is ahead of its time, that most programmers don't have the graphics or compute hardware needed to take advantage of the visualizations we have produced. It can be that programmers are too fixed in their ways and won't take the time to learn or use new tools, even if they are shown to be beneficial. It can be that today's production environments are aimed at the lowest common denominator both in terms of hardware and in terms of programmers and thus can't afford to incorporate expensive visualizations.

But it is not any of these. While they all might be a factor, advanced programmers have been using sophisticated workstations for decades and have shown great flexibility in terms of tools, languages, and environments. Moreover, today's programming environments are so feature laden that visualization tools could be incorporated without undue cost or complexity.

Others hypothesize that visualization needs to be "roundtrip", i.e. that the visualization should be tied directly to the underlying data so that the data or program being visualized can be changed by modifying the visualization [1]. While a strong case can be made for visual languages in appropriate domains, most of the applications where visualization seems promising and has the largest potential payoff, in particular program understanding, require views that are by necessity too abstract to be editable [2]. If we restrict our visualizations to views that are editable, we are losing the real power of abstraction and of visualization.

Instead, I propose that the reason that software visualization has not been successful is that almost all efforts in the field to date have and continue to be out of touch with reality. The efforts are out of touch with the reality of why we want to do visualization in the first place. The efforts are out of touch with the reality of modern software. The efforts are out of touch with the reality of software developers. To be successful today or in the future, software visualization needs to be grounded in these realities and it just isn't.

3. The Reality of Understanding

Lets first consider the motivation for doing software visualization during programming. The basic motivation is program understanding. We want to use visualization to understand our designs, our code, the structure of our code, our data structures, the execution of the code, the history of the code, and the uses of the code.

Most of today's visualization systems claim to do some form of understanding. Take my own systems for example. I used visualization in Pecan to show alternative views of the program control flow and to show dynamic execution [5]. I used static hierarchical views of the call graph and class hierarchy in FIELD to show the program structure [6,7]. Dynamic views in FIELD displayed memory utilization, resources, and file usage [8,9]. BLOOM concentrated on program traces and visualizations of data that resulted from analyzing these traces [10,11]. My most recent systems, JIVE and JOVE, show the dynamic behavior of software systems in terms of class and package usage, thread states, program phases, and basic block counts [12,13].

While these systems have made great demos and some have even been widely used, they really do not address the reality of program understanding.

Think of a program understanding problem that you have had to solve recently. For example, I have a software system that does static flow analysis of Java systems. Every now and then, generally because of the semantics of native methods or the use of reflection deep inside libraries, it fails to complete the analysis. The result is its output listing the routines that were never called or that were called and never returned contains routines that should have been called or that should have returned. To determine what is wrong in this case, I currently analyze a multiple-gigabyte trace file. What I really want is a visualization that would show us the dependencies among routines that were called and never returned or that would show blocks of code (or just the calls) that are never executed, so that I could quickly track down where the problems are and which problems are the most critical ones.

But no "visualization for program understanding" system today can even approach this problem. I could write such a system, but it would take a lot of effort and I would just throw it away once I found all the problems with the code or the libraries. Its just not worth the bother.

Another recent understanding problem I experienced was with a web crawler where multiple threads are instantiated to access and process web pages. The application runs normally most of the time, but occasionally slows to a crawl. I needed to know if this is just due to a coincidence of lots of large or slow pages, or if there is a synchronization problem in the code. I wanted a visualization that would tell me what the threads where doing in terms of the application, i.e. were they waiting for a connection, processing rotots.txt, waiting to read, parsing the page, outputting results, etc. My JIVE system could tell what state the thread was in, but this was in generic terms, not in terms of the application, and noting that a thread was doing I/O or waiting didn't really tell me what it was doing.

Another example with the same web crawler program occurred when we noted that the tag table for the HTML class was a hash table and hence all calls were synchronized. We wanted to know whether this would cause problems if we have multiple threads attempting to parse HTML simultaneously. This seems to be a good candidate for a visualization, but no visualization systems can provide this level of detailed information without significant work by the programmer.

A fourth example occurred in my software engineering class where the students were implementing the finite state logic for a 3D pinball program. In order to understand odd behaviors of their system, they needed to understand the internal state of the various automata that implement the pinball game. This is something that should be easy to visualize, but no system that I know of can provide an suitable visualization that would let the students correlate the internal state with the program as it runs.

The visualization systems that have been develop address "generic" understanding problems. They look at the program structure from the generic view of the class hierarchy or the call hierarchy. They look at dynamic understanding from the generic view of what basic blocks are being executed, what resources are being used, what is being allocated, or what states the threads are in.

But the reality of software understanding is that programmers ask specific questions, not generic ones. They want to know what state the threads of a system are in not in terms of generic states, but rather in terms of logical states from the applications point of view. They want to understand abstractions of their specific internal structures or the resultant execution as in the above example. They want to understand resource usage not globally, but rather specific to a particular event or sequence of events and then in higher level terms than the default system statistics.

I have argued and even demonstrated that systems that provide generic visualizations can be used for these specific problems. What is actually true here is that if I know what the solution to a specific problem is, I can find evidence in the generic visualization for that solution. Generic tools provide generic answers. Given the generic answer, is generally possible to guess the specific answer, but this is often difficult to do a priori and the result is a guess, not a definitive answer. Given such a guess, the programmer still has to do significant work to get the desired result. Moreover, programs are very different from one another and what is interesting on one program is not the same as what is interesting in another. Generic solutions only work for generic problems, not for specific problems in specific programs.

The reality of program understanding is that understanding involves dealing with specific problems that require program and task-specialized solutions and that software visualization has not addressed these issues.

4. The Reality of Software

The second reality that software visualization needs to address is the software itself. To be useful and incorporated into the mainstream of programming, software visualization must address the real software systems where the problems are. But what are today's software systems?

Today's systems today are large. Microsoft talks of dealing with tens of millions of lines of code in a single application. Applications of over a million lines are quite common and are typically the ones that have the most problems and where software visualization would be the most useful. Moreover these systems are dynamically complex, with large numbers of dynamically allocated objects, multiple threads of control, sophisticated user interfaces, and large libraries.

Today's systems are also structurally complex and heterogeneous. Web applications are typical today. These consist of html with embedded javascript, php, java (either as server pages, servlets, applets, or a server), SQL with a back end database system, C++ legacy systems, a variety of cgi scripts, and lots of other possibilities.

Tomorrow's systems, the ones being built today to run in the future, are even more complex. These will use web services and outside components over which the programmer has no control nor detailed knowledge. These systems will be highly distributed, running unpredictably on grids of machines, sharing data using peer-to-peer facilities, and interacting at network speeds with other, possibly outside, systems.

Software visualization systems and solutions have generally addressed yesterday's problems. They do not scale to handle today's large systems (although they now do scale to handle what were large systems a decade ago). They often do not deal with multiple threads of control even though most modern applications are multithreaded. They do not address the heterogeneous nature of today's software, instead concentrating on a single aspect or single portion of the system. They generally do not even think about how they are going to be used in understanding or analyzing tomorrow's systems. More importantly, the visualization solutions the community has developed really do not easily generalize to handle these more complex systems, and when they do, the result is often unusable

because it is overwhelming and the relevant information for a particular problem is so hard to extract.

The reality of software is that it is complex and continually growing in complexity in a variety of ways. If we want software visualization to be a practical tool for the programmer, we have to develop visualization techniques and systems that are designed to handle the complexity of tomorrow's systems, not yesterday's.

5. The Reality of Developers

The third reality that software visualization needs to address to be successful is the reality of how developers work and what they need to do their work. Visualization will be incorporated into environments when developers find it useful and demand it. But when will this be?

Software developers want to do their development. They want to do it as quickly, as accurately, and as high quality as possible. They will use whatever means they can to achieve these goals. This means that they will use new tools, languages, resources, etc. if (and this is a big if) the cost of learning that tool does not exceed its expected rewards and the tool has been and can easily shown to provide real benefits.

Software visualization has generally failed on both accounts. It is rare to find a software visualization tool that an uninformed programmer can take off the shelf and use on their particular system immediately. Most software visualization tools (many of mine own included) require the programmer to do significant work before they can receive any benefits. Some tools require extensive configuration to get a program into an environment and get it understood by the environment. Some tools require recompilation with different arguments. Some require a long program analysis process with a large database. Some require that the user work with specific languages or subsets or convert portions of the system for compatibility.

Moreover, even when the visualization tool works on a system, significant work is generally required of the programmer to tune visualization to the specific problem or task or even the particular system. For example, the programmer might need to specify the appropriate levels of abstraction for the different components in a large system in order to have the resultant display make sense. Alternatively, they might have to understand a variety of different analysis or visualization techniques to determine which makes the most sense for their particular task.

The real failure, however, is that we have not demonstrated the superiority of our detailed visualizations to much simpler techniques. Consider the case of the call graph and class hierarchy browsers that were incorporated into SGI's, Sun's, Dec's and other UNIX programming environments in the early 1990s. These provided pretty pictures and did a reasonable job of showing the program's structure. Moreover, the good ones could handle large programs through appropriate abstraction. However, in the next generation of programming tools (and still today), these were replaced by simple tree browsers that neither contain all the information of the original views nor do a good a job of explaining the nuances of the program's structure.

There are several reasons for this particular visualization failure. One is screen real estate — the graphical visualizations took up a lot more screen space without providing a significant amount of additional information. Second was how the views were used. Almost all the use of the structural views was for navigation, not for understanding [4]. If all you want to do is navigation, then the simple tree view is a much better interface for the programmer. The third reason is the configuring the graphical views for a large system took considerable programmer effort, whereas configuring a tree view is much simpler and obvious.

Showing programmers the benefits of software visualization also means dealing with the reality of addressing specific problems rather than generic ones and of handling real software systems. The bottom line here is that we have not convinced programmers that visualizations are worthwhile.

6. The Future of Software Visualization

What does all this mean for our work in software visualization. Clearly, if we as a community want to have a real impact on software development, we have to change the focus of our software visualization efforts so that they are in touch with reality. We have to think in terms of the future and not the past. We have to meet a variety of difficult and challenging goals. We have to reinvent software visualization as a solution to programmers problems rather than a generic solution looking for a problem.

We first have to comprehend what are the understanding problems that software visualization should be used to address. We shouldn't be thinking in terms of generic problems on yesterday's systems, but rather thinking about how we can make it easy for the programmer to use visualization or visualization tools to address specific problems on future systems. Moreover, we have to continually think about cost-benefit tradeoffs in any tools or visualizations we create.

With this in mind, we can enumerate some specific questions that we should be asking when we approach software visualization research or when we consider

building a new software visualization tool or system. These include:

- Does the approach scale to handle realistically sized systems today and in the future or is there some inherent limit (e.g. screen space) that is going to get in the way?
- Does the approach handle the complexities of current and future software? It makes no sense to develop visualizations today for single threaded applications when many of today's systems are and most future systems are going to be multithreaded. Similarly, one should ensure that the approach can handle distributed applications, multilingual applications, libraries, external components, databases, etc.
- Is the approach easy to use? Does the user need to understand a lot about the visualization or the visualization system in order to get anything out of the views? Here one can consider using intelligent or adaptable displays, very simple navigation and orientation techniques, and other ways of adapting the visualization to the task at hand.
- Does the visualization restrict itself to relevant information? When we draw pictures, we confine them to the information relevant to the task at hand. To answer specific problems, a visualization has to address these problems without irrelevant details. As our systems get more complex, finding the appropriate visual abstractions becomes more critical.
- Are there clear and obvious benefits of the visualization that a programmer or other developer can comprehend and appreciate? Will programmers understand how these benefits apply to their own systems?
- Does the approach let the programmer address very specific problems using the programmer's own concepts? What types of problems can it deal with and how specific can the programmer make it?
- Does the approach let the programmer address real, relevant, and important problems in software development or are the problems being solved second-order? Is visualization the best way of addressing these problems?
- Can the approach be quickly and easily customized to deal with a particular problem? Can there be a simple, easy-to-use tool that lets the programmer define the problem that needs visualization? Can we then create a visualization system that will take this problem definition and create an appropriate visualization with minimal programmer intervention?

These are the issues that we need to address if we ever want software visualization to be successful. These are the questions that we have to ask of ourselves if we want our research to be used and appreciated. These are the directions we need to follow if we want software visualization to be a viable research field.

7. References

1. Stuart M. Charters, Nigel Thomas, and Malcolm Munro, "The end of the line for software visualization?," *Proc 2nd VISSOFT*, (September 2003).

2. U. Dayal and P. A. Bernstein, "On the updatability of relational views," *Proc 4th Intl. Conf. on Very Large Data Bases*, pp. 368-377 (1978).

3. Object Management Group, *OMG Unified Modeling Language Specification*, OMG (www.omg.org) (September 2001).

4. Scott Meyers and Steven P. Reiss, "An empirical study of multiple-view software development," *Software Engineering Notes* Vol. **17**(5) pp. 47-57 (December 1992).

5. Steven P. Reiss, "PECAN: program development systems that support multiple views," *IEEE Trans. Soft. Eng.* Vol. **SE-11** pp. 276-284 (March 1985).

6. Steven P. Reiss, "Interacting with the FIELD environment," *Software Practice and Experience* Vol. **20**(S1) pp. 89-115 (June 1990).

7. Steven P. Reiss, "Connecting tools using message passing in the FIELD environment," *IEEE Software* Vol. **7**(4) pp. 57-67 (July 1990).

8. Steven P. Reiss, *FIELD: A Friendly Integrated Environment for Learning and Development*, Kluwer (1994).

9. Steven P. Reiss, "Visualization for software engineering -- programming environments," in *Software Visualization: Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc Brown, and Blaine Price,MIT Press (1997).

10. Steven P. Reiss, "Bee/Hive: a software visualization backend," *IEEE Workshop on Software Visualization*, (May 2001).

11. Steven P. Reiss, "An overview of BLOOM," *PASTE '01*, (June 2001).

12. Steven P. Reiss, "JIVE: visualizing Java in action," *Proc. ICSE 2003*, pp. 820-821 (May 2003).

13. Steven P. Reiss and Manos Renieris, "JOVE: Java as it happens," *Proc. SoftVis '05*, pp. 115-124 (May 2005).