From the Concrete to the Abstract: Visual Representations of Program Execution

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

Abstract

Programmers have always been curious about what their programs are doing, especially when the behavior is not what they are expecting. Since program execution is intricate and involved, visualization has long been used to provided the programmer with appropriate insights into program execution. This paper looks at the evolution of visual representations of program execution, showing how they have moved from concrete representations of relatively small programs to abstract representations of larger systems. Based on this, we describe the challenges implicit in future execution visualizations and methodologies that can meet these challenges.

1. Introduction

A visual representation of program execution is a graphical display that provides information about what a program is doing *as the program does it*. Visualization is used to make the abstract notion of a computer executing a program concrete in the mind of the programmer. The concurrency of the visualization with the execution lets the programmer correlate real time events, e.g., inputs, button presses, error messages, or unexpected delays with the visualization, making the visualization more useful and meaningful.

Visual representations of program execution have several uses. First, they have traditionally been used for program understanding as can be seen from their use in most algorithm animation systems [7,18]. Second, in various forms they have been integrated into debuggers and used for debugging. Finally, they have often been used as a means of doing performance analysis, visually highlighting program bottlenecks.

What makes a good visual representation depends on the particular application one has in mind. A good representation has to provide the programmer with the data relevant to the task at hand, be it understanding, debugging, or performance analysis, within the limits imposed by the display and the time constraints imposed by concurrency. Since the particular data are often not known in advance, the visualization typically needs to present as much potentially relevant information as possible, and present it in a way so that important or unusual properties stand out visually either directly or through appropriate visual patterns.

This paper is an attempt to describe what is needed to do useful visual representations of the execution of today's software, with an emphasis on understanding. We do this by looking at the different representations we and others have used in the past to learn when and how they are effective and what lessons we can draw from them. We show that the representations have been slowly migrating from the concrete to the abstract. Based on this and the needs of modern systems, we propose the use of programmer-defined abstractions as the basis for a new execution visualization framework.

2. Concrete Representations

The earliest computer-based visualizations showed the actual code as it was executed. These visualizations typically highlighted statements or lines of code as the program was executing that line. These visualization were sometimes combined with other feedback information, for example execution totals or past history.

Many of the early programming environments featured some form of dynamic visualization of the source program. For example, our PECAN environment from the early 1980's outlined the current source statement with a box [8]. This can be seen in the window at the upper right of the display shown in Figure 1. If the program was executing continually, the box kept moving around; if the program was single stepped, the box changed and the program halted with each instruction. Other dynamically updated execution views provided by PECAN included a flowchart view of the program (in the window on the lower right of the figure) and a view of the stack and the values of variables on it seen at the lower left of the figure.

PECAN was followed by algorithm animation systems such as Balsa [2,3], Tango [19], and others that included a view of the source code to highlight what the program was doing. These systems all worked because the programs under consideration were relatively small and execution time was not a primary concern.

After PECAN we tried two different approaches to handling more realistic programs. First, the GARDEN system attempted to do it using abstraction [9,11]. GARDEN was a programming system that let the user define, integrate, and use new visual languages. Each language had a graphical syntax and an execution semantics defined in terms of other languages or GARDEN primitives. Programs were represented by objects that could be

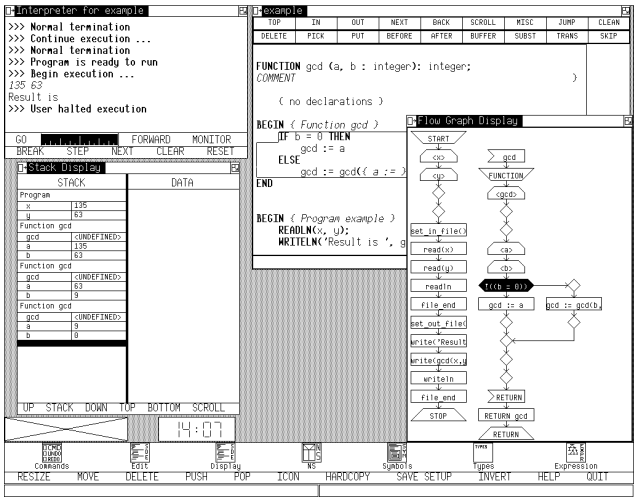

FIGURE 1. The PECAN environment run time visualization.

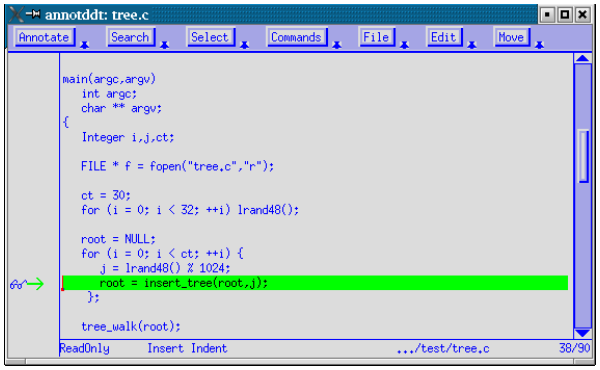

FIGURE 3. FIELD visualization showing source highlighting.

executed directly. GARDEN provided the hooks to automatically highlight execution within the visual displays of a program. Programs were typically constructed using different languages at different levels of abstraction. Since only one level of abstraction was typically displayed in a single window and the user could control the displays, the abstraction level of the visualization was effectively under the control of the user. Figure 2 shows two examples of GARDEN program visualizations, the first a flow chart view and the second a finite state automaton.

Our second approach was in the FIELD system. Here we attempted to provide visualization of full-sized C (and later Pascal, Object Pascal, and C++) systems [12,13]. While most of its visualizations were somewhat abstract (and are covered in the next section), it did source level views that updated whenever the debugger stopped execution. Moreover, it support automatic single stepping so that the user could view the program execution in the editor. This is shown in Figure 3. FIELD offered two types of source highlighting: either the text itself could be highlighted or an appropriate annotation (in this case the green arrow), would move around on the display. The editor would automatically follow execution by changing its focus or file.

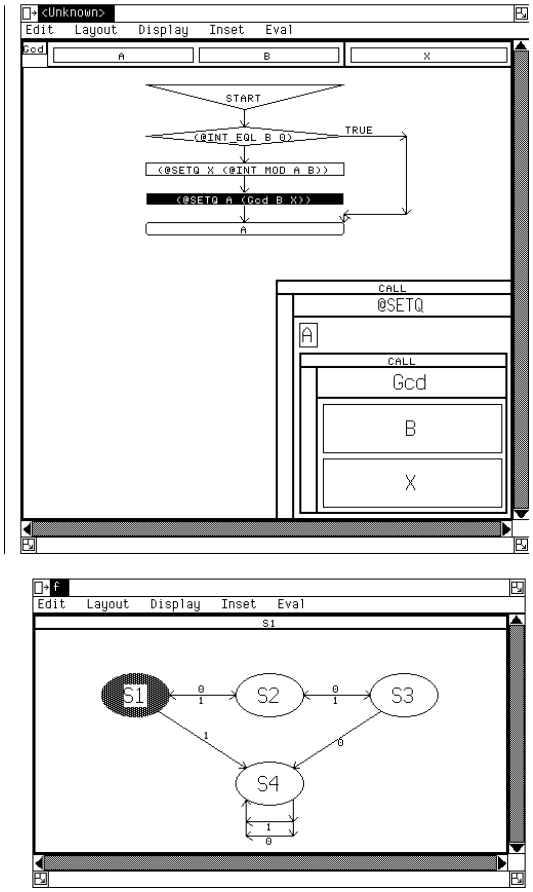

FIGURE 2. Visualization of GARDEN visual programs in action.

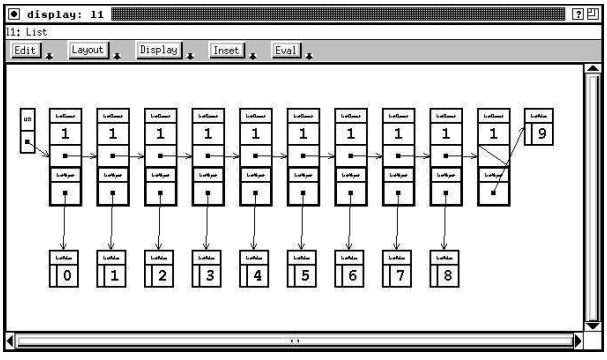

FIGURE 4. FIELD data structure display.

In addition to visualizing the source, FIELD provided visualizations of user data structures that were updated dynamically as the program executed as seen in Figure 4.

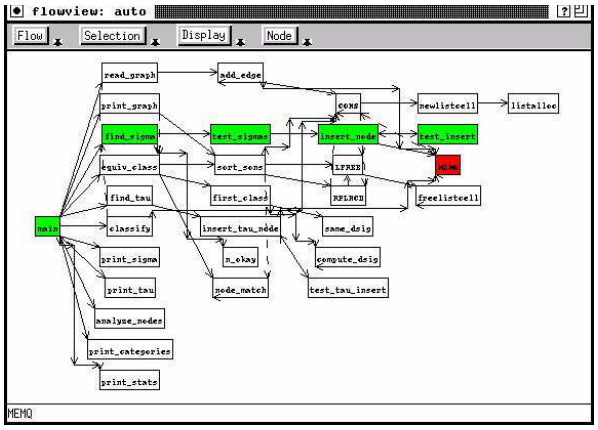

FIGURE 5. FIELD call graph visualization; highlighting shows what is currently executing.

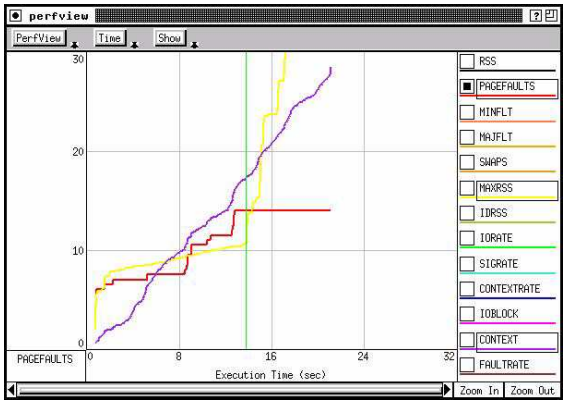

FIGURE 6. Dynamic performance visualization in FIELD.

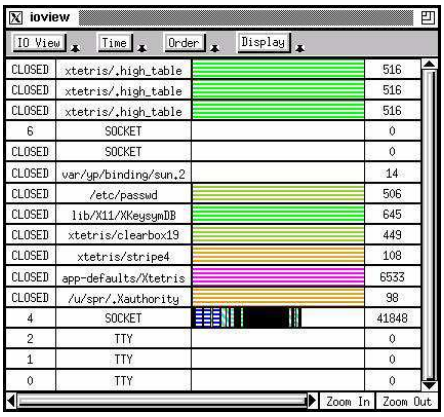

FIGURE 7. IO viewer showing file activity during execution.

The user was given control over when to update the structure to keep performance reasonable. This is similar to the displays provided by other tools [1,6] and later commercial environments from SGI and Sun. FIELD also let the user customize the data structure displays [10].

These very concrete visualizations of program execution are somewhat helpful, but found limited acceptance and practicality. Lines of code are executed much too rapidly to provide practical views of systems running at or near their normal speed. A program today can easily execute a million lines a second -- far more than can be viewed or even displayed in a meaningful way. Execution speed, when limited to that required to update the display for each line is just too slow for anything other than demonstrations or attempting to understand small programs or algorithms. What was needed was a way of viewing programs that run at closer to their normal speed.

The data structure views had other problems. First, obtaining the information needed to visualize an arbitrary data structure was costly and slowed the program down so much that the various tools updated only when the program was stopped at a breakpoint. Second, real world data structures are too complex to display in a meaningful way without significant user input.

3. Semi-Abstract Representations

Since source lines were too fine a representation to show dynamic execution, visualizations soon moved to more abstract forms. The idea here is to take a higher level view of the program and then to show the execution dynamically in terms of that view.

One obvious high-level view is that of call graphs. The FIELD environment, for example, was able to extract and display the call graph of the system in question. Execution was then shown in the graph by coloring the node currently executing and, optionally, by coloring active nodes (those on the stack) a different color. An example with the current node in red and the stack in green can be seen in Figure 5.

To handle large programs, FIELD allowed abstraction within the call graph. A node on the display could represent a single function, a file, a directory, a directory hierarchy, or, for object-oriented systems, a set of methods with the same name in multiple classes. A related view provided by FIELD showed the currently executing method via highlighting in the class hierarchy browser, a predecessor of today's UML class diagrams.

FIELD also provided visualizations that concentrated on performance and on particular behaviors. The performance view, shown in Figure 6, showed the resources that the program was using as it ran. This is closely related but more detailed than the type of views provided by operating system based visualizations such as IBM's PV [5] the visualizations that accompany MPI, or the visualizations incorporated in Sun's programming environment. Information about files and file usage during execution was shown in the file viewer seen in Figure 7. Information about memory was shown in the heap viewer seen in Figure 8. Both the file and memory views updated dynamically as the program ran.

These visualizations were more successful and useful than the earlier direct representations. The call graph and

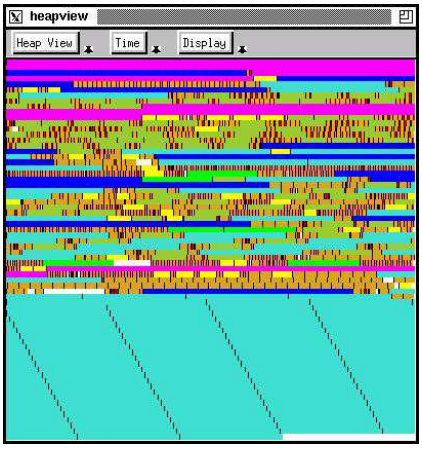

FIGURE 8. Heap visualizer showing memory utilization during execution.

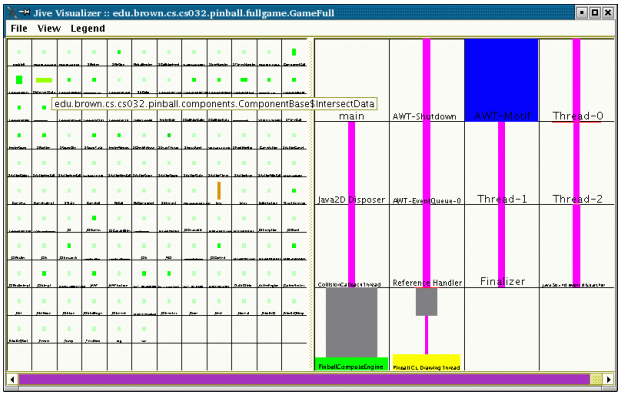

FIGURE 9. JIVE visualization. Class usage is shown on the left; thread usage on the right.

class hierarchy views were typically used by students while working on their class projects. They were not used extensively for larger systems because they did slow the execution significantly, albeit a lot less than highlighting source lines. The specific visualizations for I/O and memory found wider usage, since they could be used with minimal overhead on arbitrary systems. These visualizations were limited however by the limited domains and their lack of history. They were each aimed at specific problems such as identifying files left open or finding memory leaks, and did not extend to more general cases. While some of the motivations for a programmer to use dynamic visualization dealt with these specific problems, many others did so only partially or not at all.

4. Abstract Representations

The heap and file visualizations were successful for real systems because they allowed the program to run at or near full speed while still providing useful information. The main problem with them was that the information was quite limited in that it only touched on one particular domain and thus only helped with understanding or debugging of problems in that domain.

The reason that these views succeed was because they provided what is essentially an abstraction of the program execution. For example, the heap view built its model of memory by only looking at calls to the memory management routines; the IO visualizer did the same by looking only at file open, close, read and write routines. While these abstractions were close to the actual workings of the system, it is possible to use other abstractions to get more complete or more detailed visualizations while still maintaining program performance.

We have been working on such abstractions. Our first system along these lines, JIVE, combines several abstractions into one visualization [15]. One of these abstractions provides a view of execution in terms of classes or packages while the other provides an abstraction of thread behavior. Figure 9 show these two on the left and right respectively. JIVE runs with a slowdown factor of two.

Both of these views model the program behavior over time. The class model breaks up execution into intervals of about 10 milliseconds each. For each interval it keeps track for each class of the number of calls to methods of that class, the number of allocations done by methods of the class, the number of allocations of the class, and the number of synchronizations on objects of the class. The display then shows, for the current interval, the number of calls as the height of the bar, the number of allocations done as the width of the bar, the number of allocations of the class using the hue of the bar, and the number of synchronization as the saturation of the bar. The user can also view totals through the current interval rather than just the values of the interval and can use the scroll bar at the bottom to go back and forth in time.

The thread model on the right views each thread as being in one of eight abstract states: starting, running, running synchronized, blocking, doing I/O, sleeping, waiting, or dead. It tracks the state of each thread over time, maintaining the set of state changes and when they occur. This information can be displayed as on the right of Figure 9 as bars showing the percent of time each thread spends in each state during the current interval (or the totals up through the interval), or it can be displayed as a time graph as seen in Figure 10.

A third model of program dynamic program behavior is seen in the color of the scroll bar in the JIVE visualizations. JIVE uses the information about class and thread usage to try to match the programmer's intuition as to the phases of their program. It uses statistical methods to determine whether the current interval represents a continuation of the existing phase, a reinstatement of a previous phase, or a new phase. It then uses color to display the information about phase changes in the bottom of the window [16].

A second visualizer, JOVE, maintains a more complex model of program behavior [17]. It again looks at the program in terms of small intervals. For each interval it keeps track of how many times each basic block is exe-

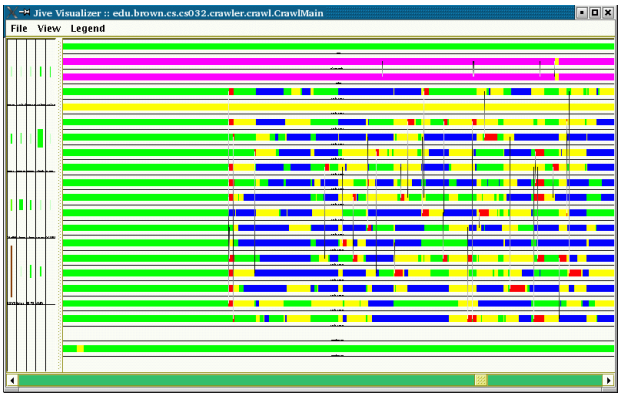

FIGURE 10. JIVE visualization showing thread states along a time line.

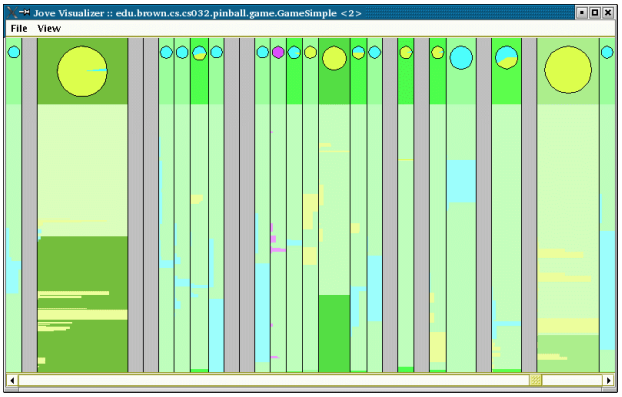

FIGURE 11. JOVE display showing thread usage at the basic block level.

cuted by each thread. The summary information is then kept over the history of the run and is used to produce displays such as seen in Figure 11. Here each vertical region represents a class. The pie chart at the top of the bar is used to show how much time each thread spent in that particular class during the interval. The height of the darker background color for the region indicates the number of allocations done. The lines within the region show information about the various basic blocks. The color of these lines indicate the thread or threads executing those blocks; the width of the line indicates the number of times the block was executed. JOVE slows the program down by at most a factor of four.

The abstract views of JIVE and JOVE are useful for providing the programmer with overview information describing what the program is doing. We have used them for debugging, understanding, and performance analysis. For the latter, they provide useful information about where execution time is spent in the program, either at a high level in JIVE or at a detailed level in JOVE. The high level view was used, for example, to determine that the 3D graphics and gravity computations of a pinball program only used about one third of the available execution time each, and hence were fast enough. The detailed view provided insights into which collision computations were the slowest.

The thread visualizations of JIVE were the most appropriate for debugging and understanding. They readily showed such events as a thread sleeping rather than waiting (and thus blocking other threads) and a thread that was blocking other threads while waiting for I/O. For a multithreaded web crawler, they showed how the threads were divided between waiting for web pages and processing the pages. They also showed the locks that occurred due to synchronization in Sun's HTML parser.

These views however suffer much of the same limits as the I/O and memory visualizations of FIELD. They address specific program aspects (albeit more general ones), and are limited to addressing issues directly related to those aspects. They provide general information about the program execution rather than information that is specific to the particular application or the coding abstractions.

5. Programmer-Defined Representations

The challenge for dynamic program visualizations is to provide information that is meaningful for understanding specific but not yet defined questions while running the application at or near full speed.

Our experiences show that the visualizations that have been most widely used and appreciated for production programs are those that provide a visual model of some aspect of the execution and dynamically update that model as the program runs. These include the memory and I/O visualizers of FIELD and the thread visualizers of JIVE. These systems worked because the model they provide is directly relevant to both the program and to particular problems that are of interest to the programmer. While it is difficult to get a gestalt of the memory behavior of a program from a typical debugger or print statements, the memory visualizer provides such a view at a glance. Through visual patterns it quickly shows memory leaks, abnormally large or unusual allocations, and memory fragmentation. The thread visualizations do similar things related to problems relevant to thread behavior and interaction.

If such views are going to be extended to make dynamic visualization more useful in general, they will have to be based on models that address the issues that programmers want to understand or debug about their particular systems. These models will need to reflect how programmers view their systems, dynamically update these models as the program runs, and then provide visualizations of these models that convey the necessary information.

Such models can be program or language specific. For example, consider a multithreaded web crawler. Each thread repeatedly is assigned a page. It reads that page, parses it, computes summary information, and then stores data about the page based on the parse. Programmers might want to see what each thread is doing in terms of this

model. They want to differentiate parsing the page from computing the summary information; they want to know when it is waiting for the web versus waiting to write information to disk. Essentially, they have a model of thread behavior and want to see a visual display of that model.

As a simple example of language-based models, consider iterators in Java. Suppose one wants to track all the currently active iterators in a program, seeing which are currently active, and ensuring each is used correctly, e.g. that *hasNext* is called before *next*. A dynamic visualization could provide a display that showed each active iterator as a box colored by its current state (unused, *hasNext* called, *next* called, *next* called without *hasNext*), with positional information relating the iterators to the source or to particular threads. From such a display the programmer would see what is going on and potential errors would stick out.

These and other visualizations can be provided by letting programmers define the appropriate models for their programs and then providing suitable visualizations. To be practical, the models must be easy to specify and understand, quick to do, and reusable. The set of visualizations provided must be flexible and easily adaptable to the different models. The challenges here involve finding the right framework for defining the models, defining an appropriately broad set of visualizations, providing support to automate or simplify associating the model with the visualization, ensuring that the models can be derived from program execution with small overhead, and doing all of this in a visual framework.

These challenges can be met. Automata over parameterized program events combined with suitable data structures can be used to model the above situations with minimal overhead and with a visual language front end. Multiple visualizations can be defined and associated with models using appropriate heuristics as in [14]. Systems such as Aspect/J show that program events can be detected efficiently [4].

6. Conclusions

Visualizations of program execution have evolved from concrete representations of the source code that were slow and only practical for simple programs, to abstract representations that, while they don't show everything, show detailed information about some aspect of the execution and that work for real systems. To extend the utility of such dynamic visualizations, one needs to look at maintaining and visualizing new abstractions as the program runs. We propose a model whereby programmers can easily define such abstractions that are relevant to their particular understanding or debugging tasks and then have appropriate visualizations generated from these abstractions. This is the next phase of dynamic program visualization.

Acknowledgements. This work was done with support from the National Science Foundation through grants CCR021897 and ACI9982266.

7. References

1. David B. Baskerville, "Graphic presentation of data structures in the DBX debugger," UC Berkeley UCB/CSD 86/260 (1985).

2. Marc H. Brown and Steven P. Reiss, "Debugging in the BALSA-PECAN integrated environment," ACM SIGPLAN-SIGSOFT Symposium on Debugging (1983).

3. Marc H. Brown and Robert Sedgewick, "A system for algorithm animation," *Computer Graphics* Vol. **18**(3) pp. 177-186 (July 1984).

4. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An Overview of AspectJ," in *European Conference on Object-Oriented Programming*, (2001).

5. Doug Kimelman, Bryan Rosenburg, and Tova Roth, "Visualization of dynamics in real world software systems," pp. 293-314 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price,MIT Press (1998).

6. Brad A. Myers, "Displaying data structures for interactive debugging," Xerox csl-80-7 (June 1980).

7. B. A. Price, I. S. Small, and R. M. Baecker, "A taxonomy of software visualization," *Journal of Visual Languages* Vol. **4**(3) pp. 211-266 (Dec. 1993).

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

9. Steven P. Reiss, Eric J. Golin, and Robert V. Rubin, "Prototyping visual languages with the GARDEN system," *Proc. IEEE Symp. on Visual Languages*, (June 1986).

10. Steven P. Reiss and Joseph N. Pato, "Displaying program and data structures," *Proc. 20th Hawaii Intl. Conf. System Sciences*, (January 1987).

11. Steven P. Reiss, "Working in the Garden environment for conceptual programming," *IEEE Software* Vol. **4**(6) pp. 16-27 (November 1987).

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

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

14. Steven P. Reiss, "A visual query language for software visualization," *IEEE 2002 Symposium on Human Centric Computing Languages and Environments*, pp. 80-82 (September 2002).

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

16. Steven P. Reiss, "Dynamic detection and visualization of software phases," *Proc. Third International Workshop on Dynamic Analysis*, (May 2005).

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

18. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, *Software Visualization*: *Programming as a Multimedia Experience*, MIT Press (1998).

19. John T. Stasko, "TANGO: a framework and system for algorithm animation," *IEEE Computer* Vol. **23**(9) pp. 27-39 (September 1990).