Visual Representations of Executing Programs

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

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

1. Introduction

An on-line visual representation of an executing program 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 in conjunction 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 executing programs have several uses. First, they have traditionally been used for program understanding as can be seen from their use in most algorithm animation systems [37,52]. Second, in various forms they have been integrated into debuggers and used for debugging [2,31]. Finally, they have often

been used as a means of doing performance analysis, visually highlighting program bottlenecks or abnormalities and correlating them to what is happening in the environment [24,29].

What makes a good visual representation depends on the particular application that 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.

While this paper concentrates on visualizing executing programs as they execute, we note that there has also been a significant body of related work that looks at visualizing and understanding the dynamic behavior of software by capturing program traces while the program is executing and then visualizing these after the program has completed. This off-line approach is sometimes considered more appropriate for tasks such as reverse engineering where little is assumed about a program and where the task involves attempting to achieve an overall understanding of what happens during execution and how it correlates with the source [32,54].

On-line and off-line visualizations are similar and quite different at the same time. Most of the graphical representations that are used in on-line visualizations also appear in some set of off-line visualizations. The inverse is not always true

because off-line visualizations can afford to be more computationally intense either in terms of the data analysis needed to put up the visualization or in terms of the graphics needed for the visualization. Both types of visualizations typically view execution as a sequence of events. For off-line these events are recorded and analyzed later; on-line visualization requires any analysis be done as the events are generated. This restricts the types of analyses that can be done and puts a heavier emphasis on limiting the set of events or on doing the necessary analysis before generating events. Some off-line techniques have looked at simplifying the instrumentation, but this has not had the same importance as it has with on-line visualizations. Both types have struggled over the years to allow the visualization of larger and larger programs. Again, the limitations imposed by on-line visualization have yielded a different set of solutions, with more emphasis on limiting events and simplifying graphics rather than on detailed analysis of the event streams.

Both on-line and off-line visualizations have their place. When a detailed analysis of one or several runs is needed to understand overall program behavior such as during reverse engineering, the advantage of having trace data and thus being able to do several different analysis makes off-line visualization beneficial. However, when looking for a specific problem that arises only occasionally as in debugging, an on-line approach is generally easier to use. Where programmers are looking at their own software and have a good understanding of what should be happening, on-line visualizations typically can provide significant interesting information without the overhead of collecting traces and doing the various analyses.

This paper is an attempt to describe what is needed to do useful visual representations of today's software as it executes, with an emphasis on understanding the execution as it happens from different perspectives. 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 each line. These visualization were sometimes combined with other information, for example data values, execution totals or past history.

2.1 Visualization in Early Programming Environments

Many of the early programming environments featured some form of on-line dynamic visualization of the source program. For example, our PECAN environment from the early 1980's outlined the current source statement with a box [38]. This outlining 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

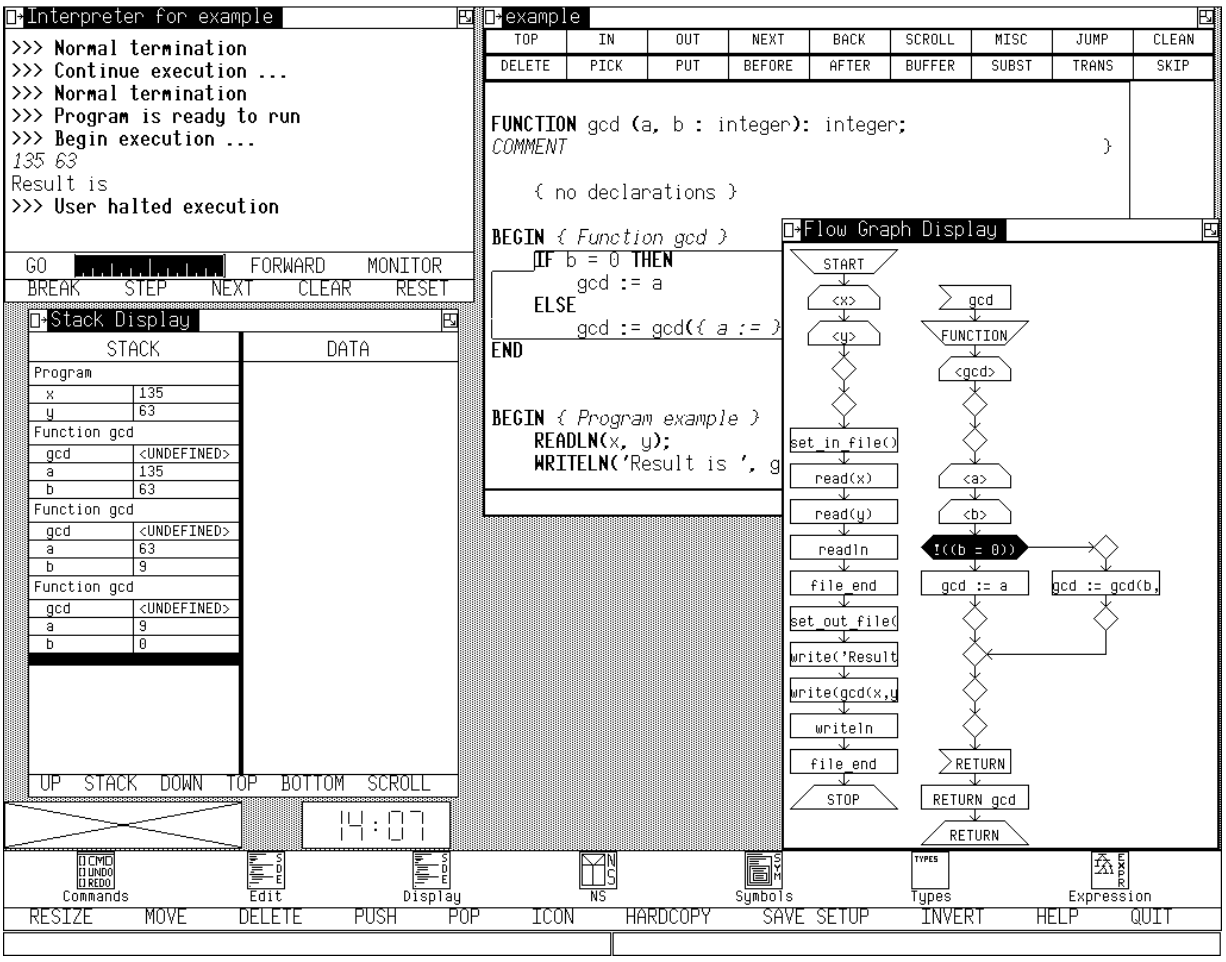

FIGURE 1. The PECAN environment run time visualization.

view of the stack and the values of variables on it (seen at the lower left of the figure).

While the PECAN visualizations were fun to look at, they were not that practical. First, PECAN was designed as a prototype system and could only handle relatively small programs (under two thousand lines). Thus, one never really tried it with real systems. Second, while it was designed to support multiple languages, the working implementation only supported a simple, interpreted version of Pascal, further limiting the potential applications and use. The visualizations themselves were simulta-

neously both too fast and too slow. They were too fast to understand because they updated faster than the eye could focus. They were too slow because doing the visualization slowed the program down substantially so that it was impractical to use for anything complex. However, the visualizations did seem appropriate for teaching and understanding the basic execution of programs. Since these uses were not in the environment's target domain, we did not evaluate this particular application.

Simple code-based visualizations in programming environments such as those in Pecan have been duplicated over time. Lieberman's ZStep 95 provided a source-oriented execution view which such highlighting but also saved the execution states to allow the user to scroll back and forth in time over the visualization [26]. Most current programming environments highlight the current line as the debugger traces or stops.

2.2 Algorithm Animation

Teaching and simple program understanding, however, was the focus of the various algorithm animation systems that were developed starting at about the same time as PECAN. Systems such as Balsa [5,6], Tango [53], and others typically included a view of the source code to highlight what the program was doing in addition to their characteristic algorithm-specific visualizations. These systems all worked because the programs under consideration were relatively small and execution time was not a primary concern. Indeed, Balsa even included a user interface control that let the programmer slow the program execution down so that they could examine it in slow motion.

What the algorithm animation systems provided, beyond the simple programming environments, was views of the data. Here they provided very specialized views that were more geared to the algorithm in question than to the specific program data structure. For example, for sorting, instead of displaying the array containing the data, they might display a matrix which shows data position over time where time is counted in the number of data exchanges [1]. Alternatively, they might display an array used as a heap as the corresponding tree.

This contrasts to the simple data displays that were provided by PECAN which only showed the stack. Arrays, structures, and pointers on the stack could be explored by explicitly clicking on them to expand them, but the resultant display was shown as a tree and didn't really illustrate how complex data structures were represented in memory.

2.3 Visualizing Visual Programming

After PECAN we tried two different approaches to handling more realistic programs. The first, the GARDEN system, attempted to do it using conceptual programming and visual languages [39,40]. GARDEN was a programming system that let the user define, integrate, and use new visual languages, with the idea being that users could conceptualize their problem in terms of some visual language, use GARDEN to implement that language, and then easily code up their particular problem. 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 executed directly. GARDEN provided the hooks to automatically high-

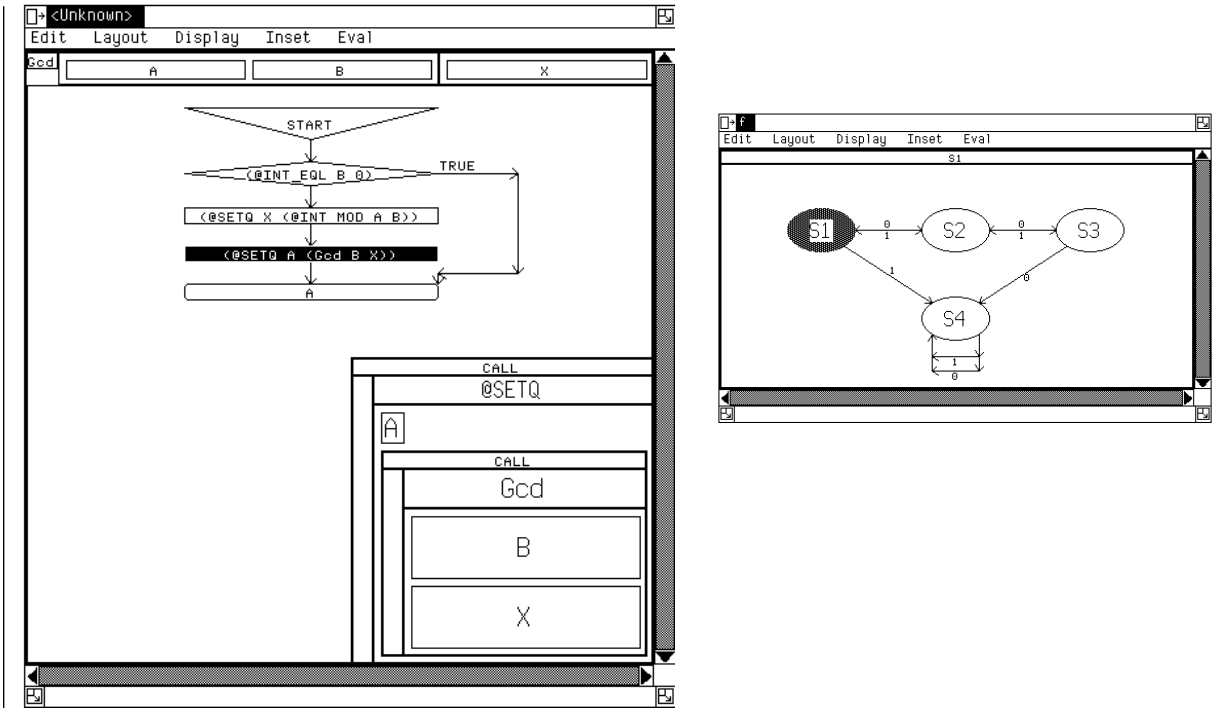

FIGURE 2. Visualization of GARDEN visual programs in action.

light execution within the visual displays of a program to provide on-line visualization. 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 flowchart view and the second a finite state automaton. The flowchart displays the overall flowchart in the center and a boxed view of the currently selected node in the lower right-hand corner. The finite state automata displayed label states and arcs. In the flowchart, execution was shown by highlighting the current node. With finite state automata, the current state and traversed arcs were highlighted. Other languages that were implemented

included Petri nets, data flow graphs, query-by-example, statecharts, and port-based modules.

Because the focus of GARDEN was on visual languages and program design, the visualizations that were provided were somewhat incidental and the program size remained relatively small. Again, while GARDEN was used for a variety of experimental applications, its visualization package saw little use. The reason again was speed. The visualizations only worked when GARDEN was interpreting the visual program. When the program was running at full speed, the visualization was a blur and was not particularly helpful. Moreover, when programs were compiled to be run for production, the visualization could not be used. The real utility of the visualization was when debugging GARDEN programs, when the visualization would provide the user with the proper context within the visual program.

Many of the other systems that supported visual languages also provided some sort of dynamic view of the program in terms of the language. Early examples include Kimura's data flow language Show and Tell [25], Glinert's flowchart-based Pict [17], and Jacob's state-transition language [21]. One could also consider the immediate feedback provided by systems such as Borning's Thinglab [4] as a graphical view of an executing program, although in this case the view is actually the output.

2.4 Source and Data Visualizations

Our second approach to handling more realistic programs was in the FIELD system. Here we attempted to provide on-line visualization of full-sized C (and later

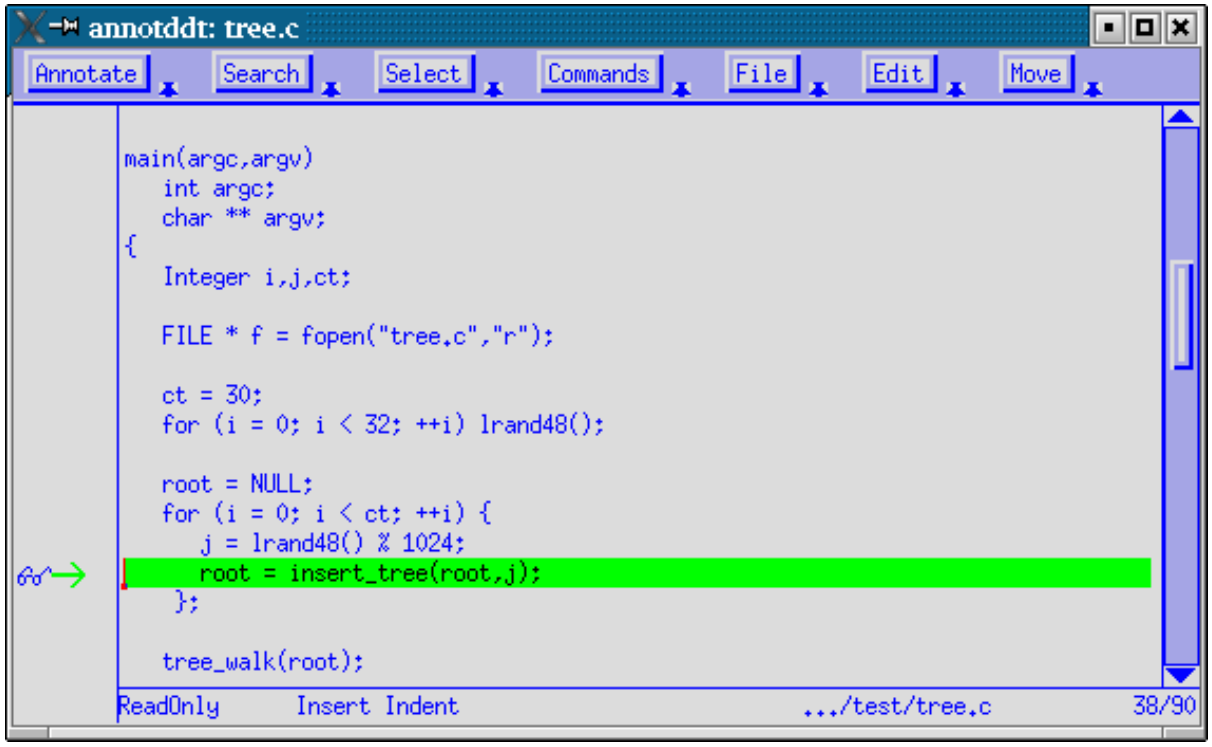

FIGURE 3. FIELD visualization showing source highlighting.

Pascal, Object Pascal, and C++) systems [42,44]. While most of its later visualizations were somewhat abstract (and are covered in the next section), the early visualizations offered source level views that updated whenever the debugger stopped execution. Moreover, it supported automatic single stepping so that the user could view the program execution in the editor. This feature is shown in Figure 3. FIELD offered two types of source highlighting: either the text itself could be highlighted or an appropriate annotation would move around on the display. The example in the figure shows both. The current line of text is highlighted with a green (dark) background and the arrow annotation on the left hand side points to the executing line. The editor would automatically follow execution by changing its focus or file as the program ran.

Because the visualization was implemented by continually (but automatically) single stepping the debugger, the performance was about right for program understanding, but was too slow to use continually on a real system. To get around this limitation, FIELD provided mechanisms to restrict the single stepping and hence the visualization to selected functions, files, or classes. The result was a source-based visualization that was actually used both on advanced programs and on student programs, albeit for limited applications.

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. The user was given control over when to update the structure to keep performance reasonable. These displays were similar to the displays provided by other tools [2,30] and later commercial environments from SGI and Sun. FIELD went beyond these other tools by letting the user customize the data structure displays using the same visual editor that was used for defining visual languages in GARDEN [41]. The customized displays let the user abstract their data structure and view it in their terms. This customization can be seen in the simplified list visualization of the data structure at the top of the figure shown in the middle or the display of a linked tree structure displayed as a tree at the bottom of the figure.

The data structure display looked nice but actually had limited utility other than for debugging. The main problem was the cost of updating the diagram. In order to get the information needed to display or update the diagram, FIELD needed to interact extensively with the debugger. Such interactions were expensive and could only

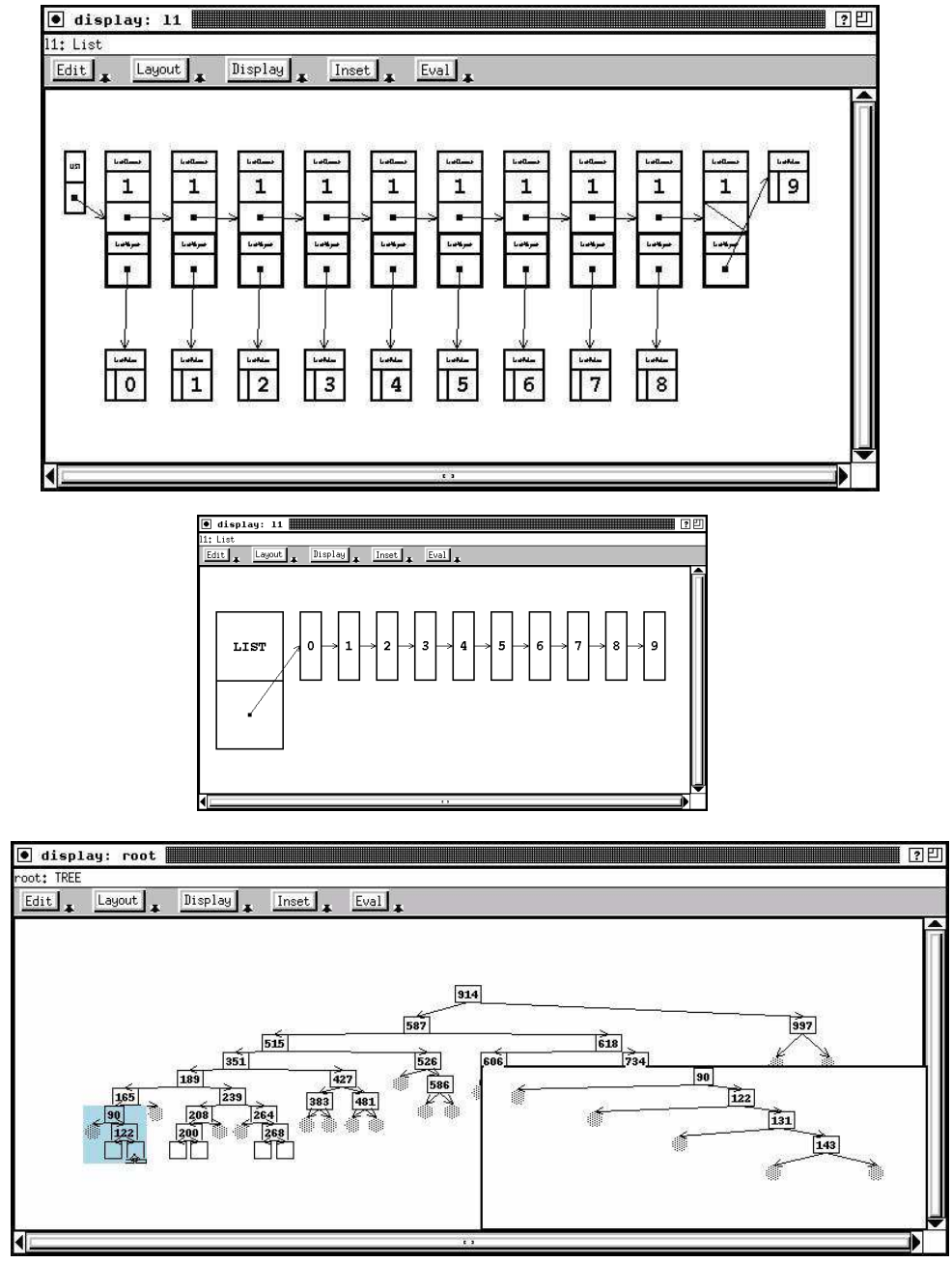

FIGURE 4. FIELD data structure displays.

be done practically when the program stopped execution. Even then, it took a matter of seconds and so wasn't always desirable. The second problem was that real world data structures were just too complicated to display even with significant user customization. The third problem was that the visual editor used to define the custom

visualizations was relatively complex and the time commitment needed to create a pleasing data structure display was not worth it for ephemeral applications.

2.5 End-User Programming Systems

Spreadsheets such as VisiCalc introduced the notion of wysiwyg programming, i.e. the notion that the results of the program, including intermediate results, are continuously computed and shown to the programmer while the program is being developed. By their very nature, such systems provide a concrete view of the execution of the underlying program.

Early work on generalizing the simple spreadsheets yielded systems like XED or VisiProg [20,22]. This system offered a simple procedural language with an underlying data flow model. Programs were continually executed as they were written and the output was continually shown as part of the environment.

This work has generalized in two directions. First, the notion of continuous testing has been extracted to provide a simple high-level visualization of a program — whether its test cases currently succeed or fail [50]. Second, as spreadsheet systems have become more powerful, they have become real programming languages in their own right.

Visualizations that have then been built on top of spreadsheet systems are inherently dynamic. For example, the work of Burnett, et al. shows how one can visualize recursive spreadsheet programs [7], assertions about spreadsheet nodes [8], and test cases and the inherent correctness of nodes [49].

3. Semi-Abstract Representations

The primary drawback of the concrete representations cited above was their inability to show real programs in action. There were always performance issues, either with the program running too slowly because of the graphics or the graphics running too quickly to be understood. A large part of the underlying problem was that source lines and actual user data structures are too fine a representation to show dynamically. Because of this, practical run time visualizations moved to more abstract forms. The idea here was to take a higher level view of the program and then to show the execution dynamically in terms of that view.

3.1 Function-Level Visualizations

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 [42,43]. On-line 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 (dark gray) and the stack in green (light gray) can be seen in Figure 5. The colors here were selectable through resource files. 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. Moreover, nodes could be selectively eliminated in order to simplify the display. The visualization here was much more practical than the earlier line-

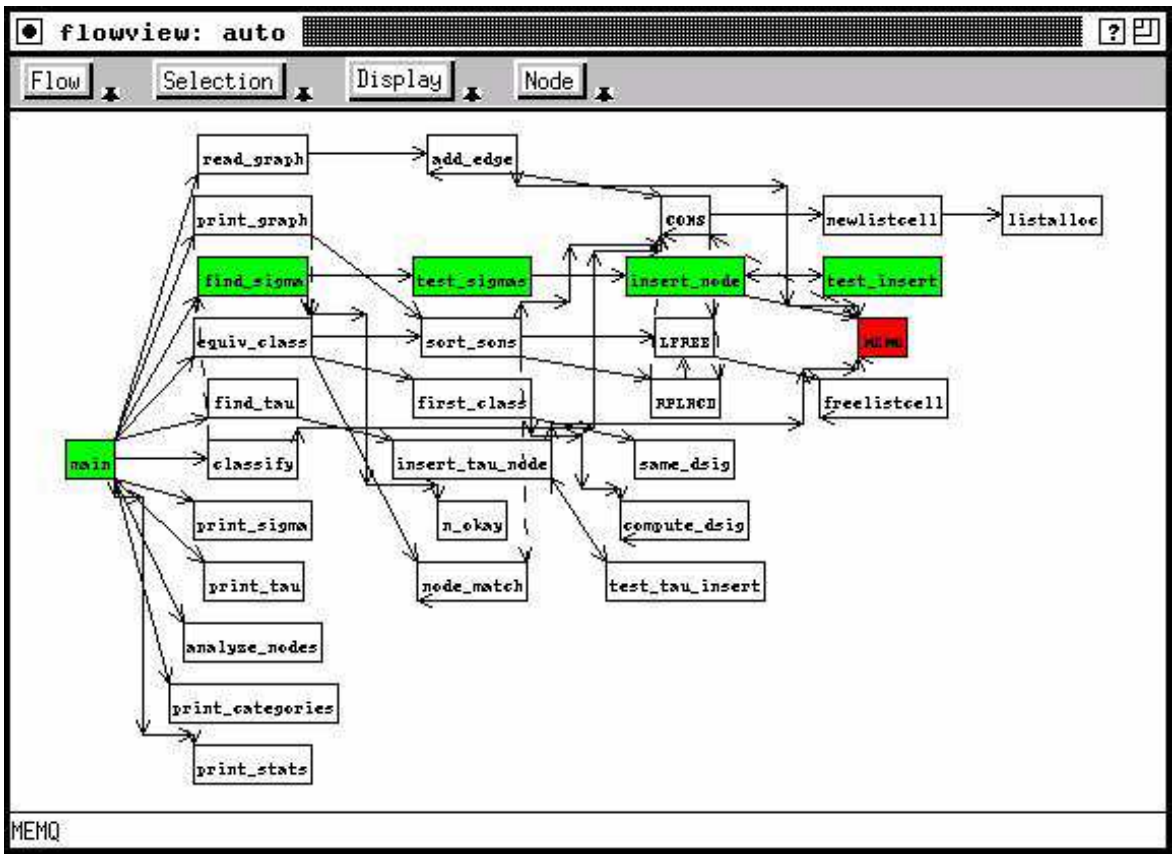

FIGURE 5. FIELD call graph visualization; highlighting shows what is currently executing.

level displays. The display could be set to update dynamically or only when the program stopped execution. Using dynamic display did slow the program down significantly because it used tracing within the debugger to trigger the display. However, the slow down was not so much as to make the program unusable. As a result this type of visualization was used to class to demonstrate the execution of various programs and by students to achieve an understanding of their own program's execution.

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. A simple view from this browser can be seen in Figure 6. The class browser

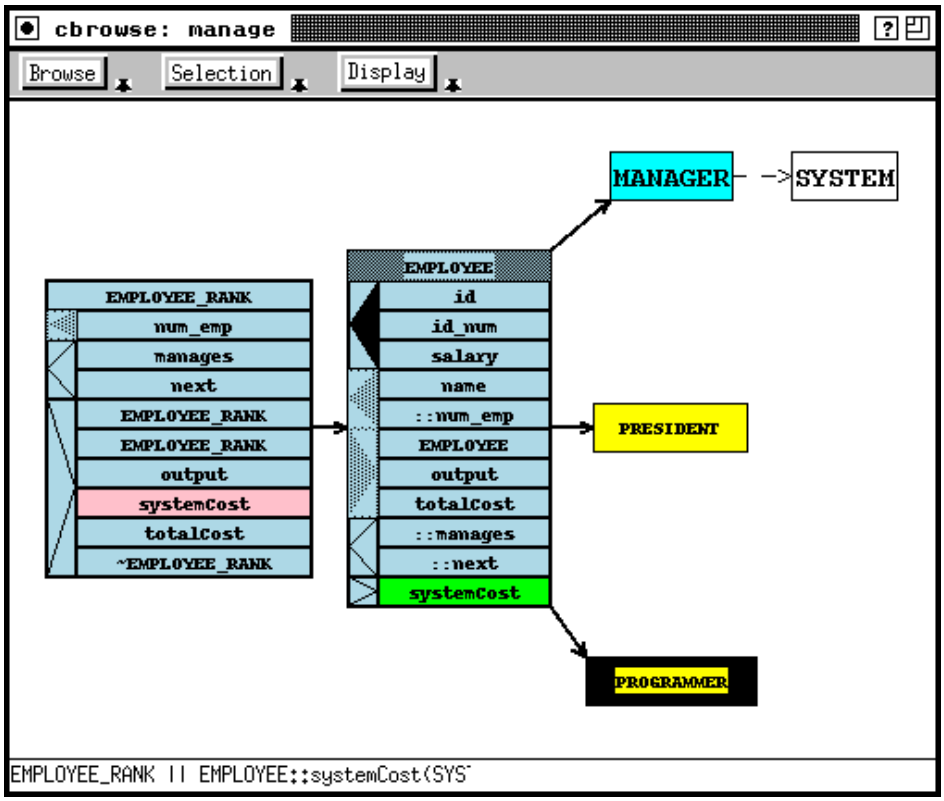

FIGURE 6. The FIELD call graph browser also showed dynamic information.

concentrated on displaying the class hierarchy and other relationships among classes and letting the user interactively simplify or specialize the class diagram. Dynamic visualization was shown by highlighting the currently executing method using the same underlying facilities as with the call graph browser. In this case only the current method was highlighted and no attempt was made to show the call stack or non-method routines. The execution visualization provided by this view was used extensively by the students for debugging and to a more limited extent for program understanding.

3.2 Specialized Visualizations

FIELD also provided on-line visualizations that concentrated on performance and on particular behaviors. The performance view, shown in Figure 7, showed the

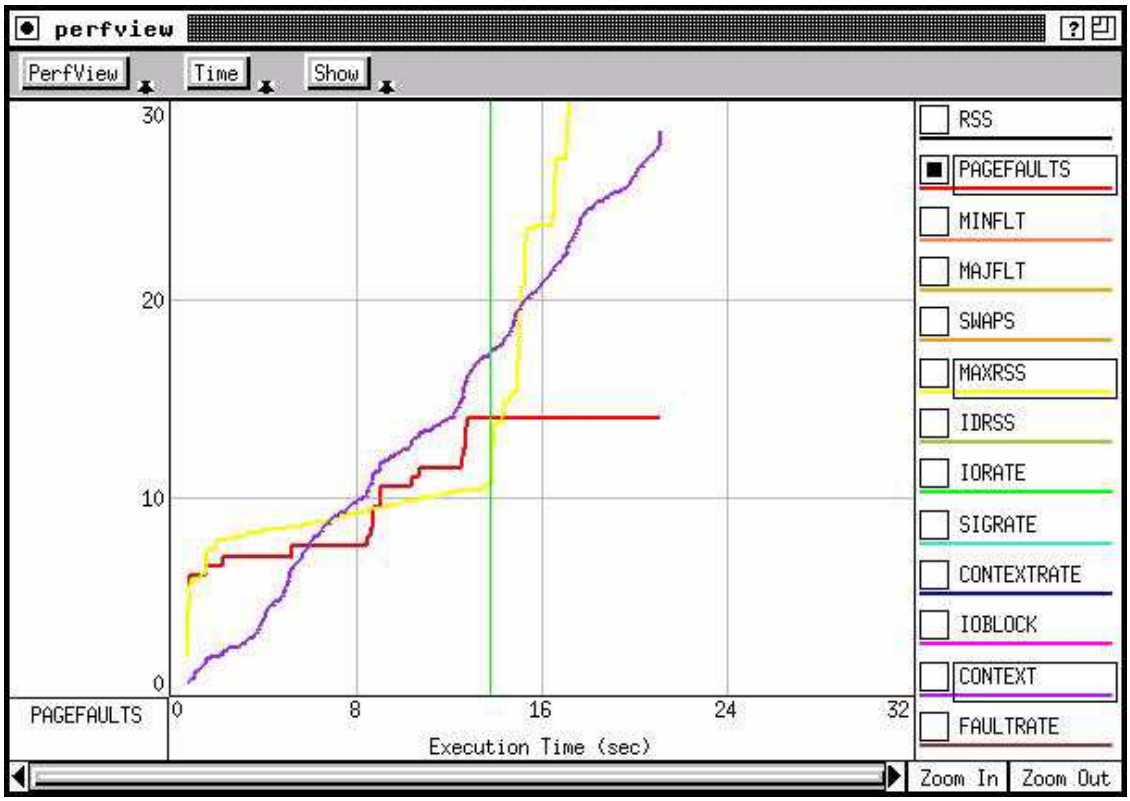

FIGURE 7. Dynamic performance visualization in FIELD.

resources that the program was using as it ran. This view was supported by a little library that would periodically wake up and send the relevant program statistics to the visualizer. Its utility was limited by the lack of accuracy of the Unix resource system calls at the time.

There have been a wide variety of different dynamic performance visualizations. Many of these concentrated not on the user's application, but on system performance in general while the application was being run. For example, IBM's PV [24] system provided a variety of different operating-system level performance visualizations that could be used to gain insight in the actual application. Another set of performance visualizations was tied to the behavior of parallel applications and how they made use of their resources. The best examples of these are the visualizations that

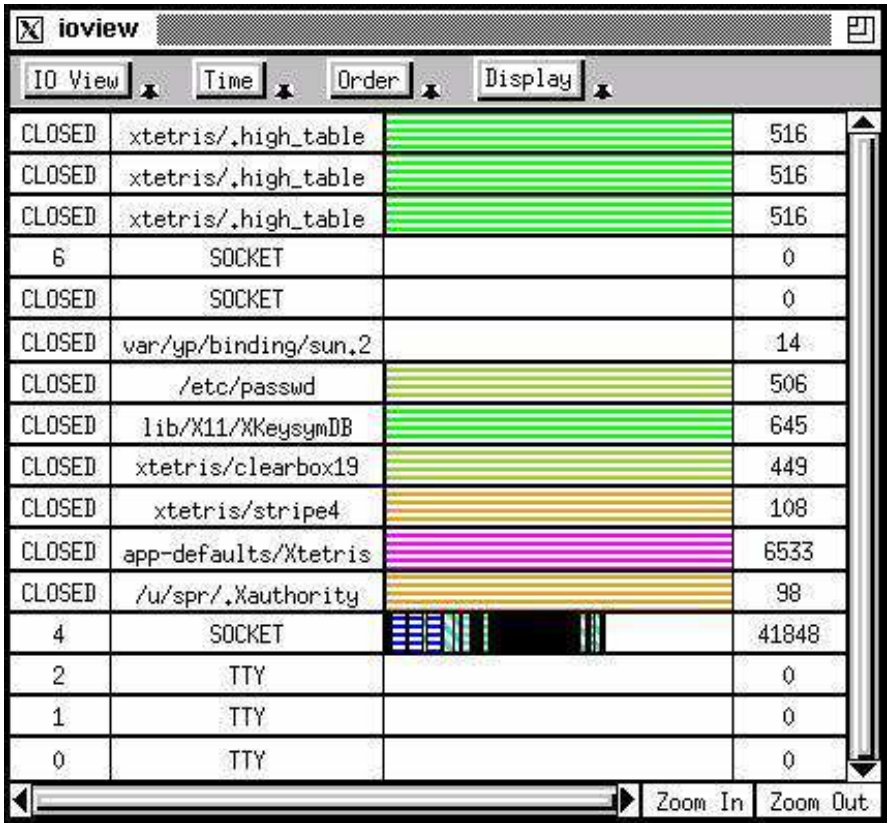

FIGURE 8. I/O viewer showing file activity during execution.

accompany MPI such as XMPI [9] or MPE [11]. More recently, environments such as Sun's Studio incorporate tools that provide visualizations of gprof [18] data that update as the program runs [29].

Information about files and file usage during execution was shown in the file viewer seen in Figure 8. This view tracked every file operation and provided a graphical display showing the results. Color could be used to either indicate the size of the I/O operation or the time the I/O operation was done. Reads and write were distinguished by horizontal versus diagonal lines respectively, and the position of the operation in the vertical bar showed its location in the file. This view was very helpful for finding potential file errors, for example files that were left open, files that were

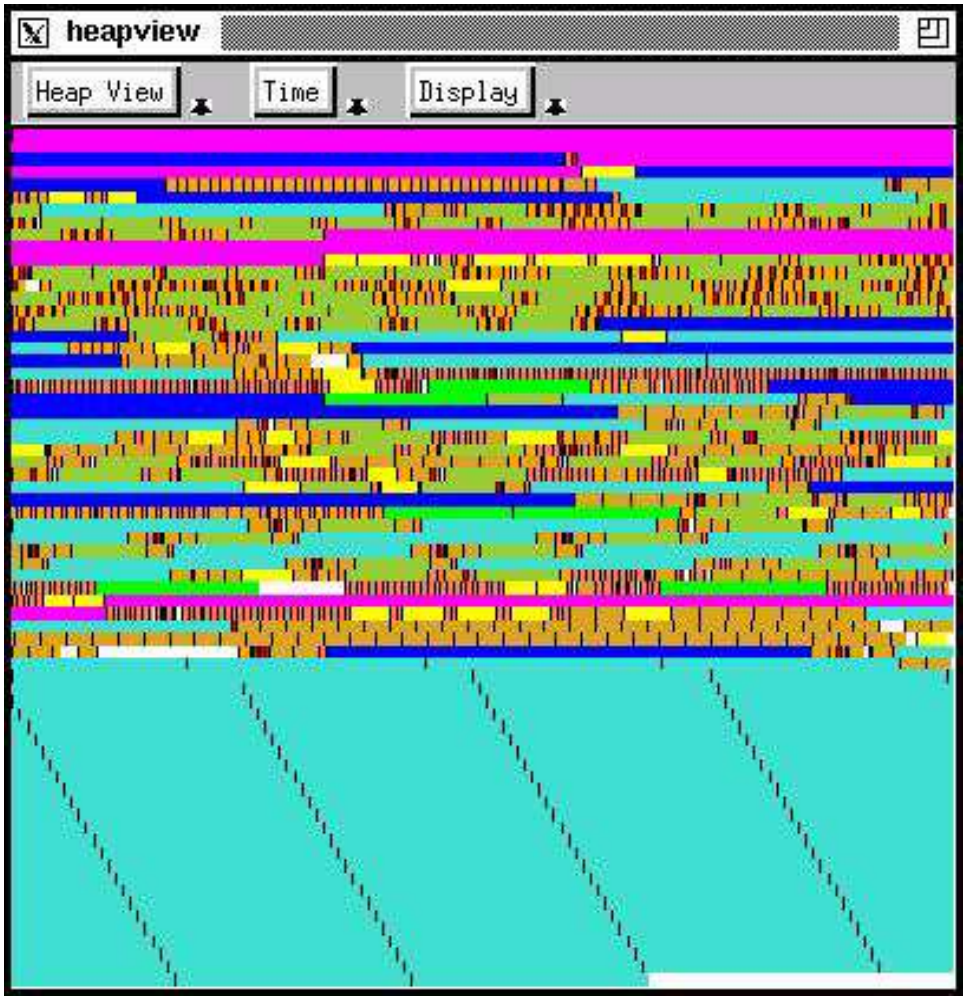

FIGURE 9. Heap visualizer showing memory utilization during execution.

opened multiple times, and files with lots of little I/O operations. However, since these situations arise relatively infrequently, the visualization had limited utility.

Information about memory was shown in the heap viewer seen in Figure 9. This visualization was driven by tracing all calls that allocated or freed memory and sending the parameters and return values of those calls to the visualizer. Because the amount of data involved is relatively small, this visualization was able to run in real time with the program and only slowed the program by a small factor. The visualization itself was also quite helpful in detecting several types of memory problems

that affected C and C++ programs. Color in the visualization could indicate size, time, or the source of the allocation. Coloring let the view be used to show at a glance excessively large or small allocations, allocations that were supposed to be ephemeral but weren't, the allocations that arose from executing a particular command, or allocation hot spots in the underlying system. Moreover, the overall visualization gave a good view of how memory was used, how fragmented it was, and quickly illustrated such problems as the memory leak shown at the bottom of Figure 9 by the area that just kept growing during execution.

While these specialized visualization were much more widely used and more generally useful on real systems, they still had their drawbacks. They were restricted by their limited domains and their lack of history. They were each aimed at specific problems such as identifying files left open or finding memory performance issues, and did not extend to more general cases. Moreover, while they showed what was happening now, they did not let the programmer go back and explore what happened in the past to get to this point. While some of the motivations for a programmer to use dynamic visualization dealt with the specific problems that these visualizations addressed, many other motivations were handled only partially or not at all.

3.3 Non-Procedural Visualization

The above visualizations were geared toward standard programming languages and environments. Other approaches to programming required different approaches to visualization.

One such approach is seen in the Transparent Prolog Machine [14] and the successor system MRE. These systems provide a visualization of Prolog execution that have been used for debugging, understanding the semantics of Prolog, and program understanding [15]. The visualizations show what prolog is doing but provide a degree of abstraction by presenting the execution in terms of trees and allowing simplifications of the trees. The visualization could be run on-line or off-line with replay capabilities.

Just as Prolog execution can be difficult to understand, so can rule-based (knowledge-based) systems. These take a set of rules and a set of features and trigger rules based on the current values of features, with rules then changing the feature set. There have been several visualizations that provide the user with an understanding of the execution of such systems [12].

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 problems in that domain.

The reason these views succeeded 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 I/O visualizer did the same by looking only at the file open, close, read and write rou-

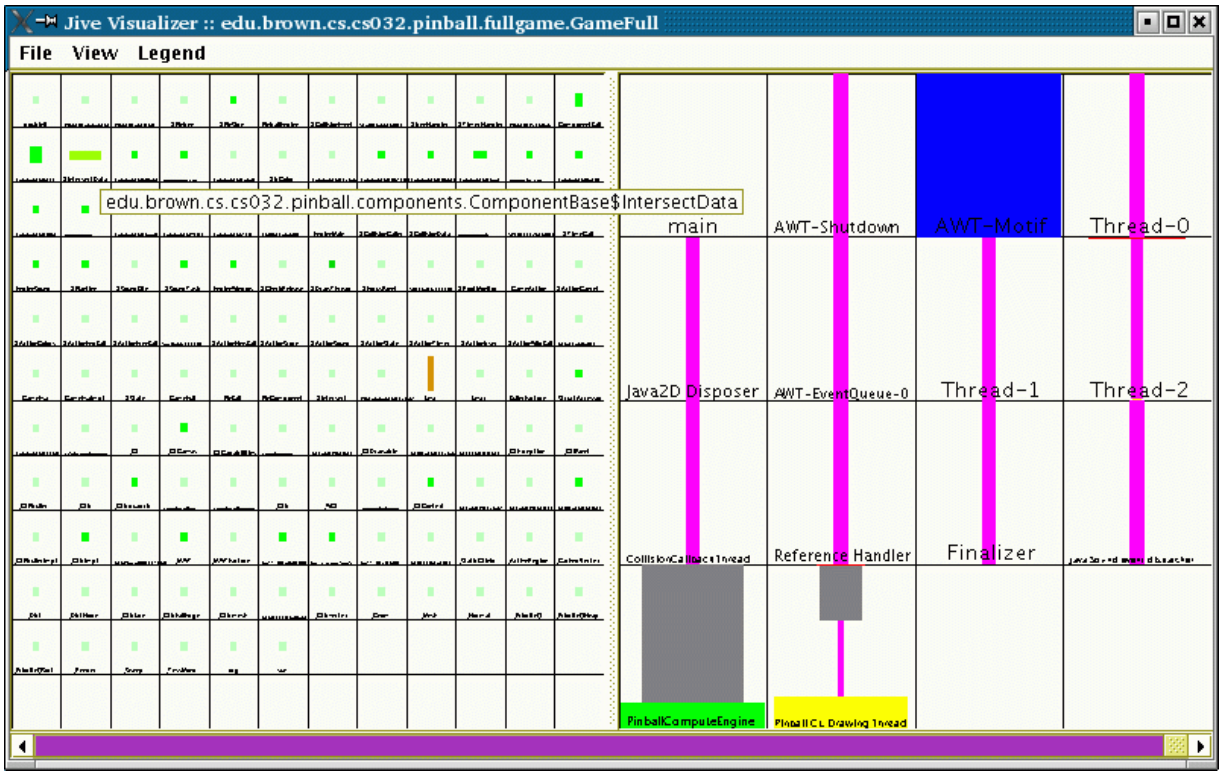

FIGURE 10. JIVE visualization. Class usage is shown on the left; thread usage on the right.

tines. 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.

4.1 JIVE

The desire to provide more useful on-line visualizations of program execution led us to take a new approach that emphasized abstraction and minimal instrumentation. Our first system along these lines, JIVE, combines several abstractions into one visualization [46]. 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 10 show these two views on the left and right respectively.

Both of these views model the program behavior over time. The class model breaks up execution into intervals of about ten 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. This view provides the user with insight as to where execution is occurring, whether there are excessive allocations, and where synchronization is being used.

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 seen on the right of Figure 10) 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 11). In the latter case, we were also able to illustrate synchronization dependencies between the threads using vertical lines that went from white to black.

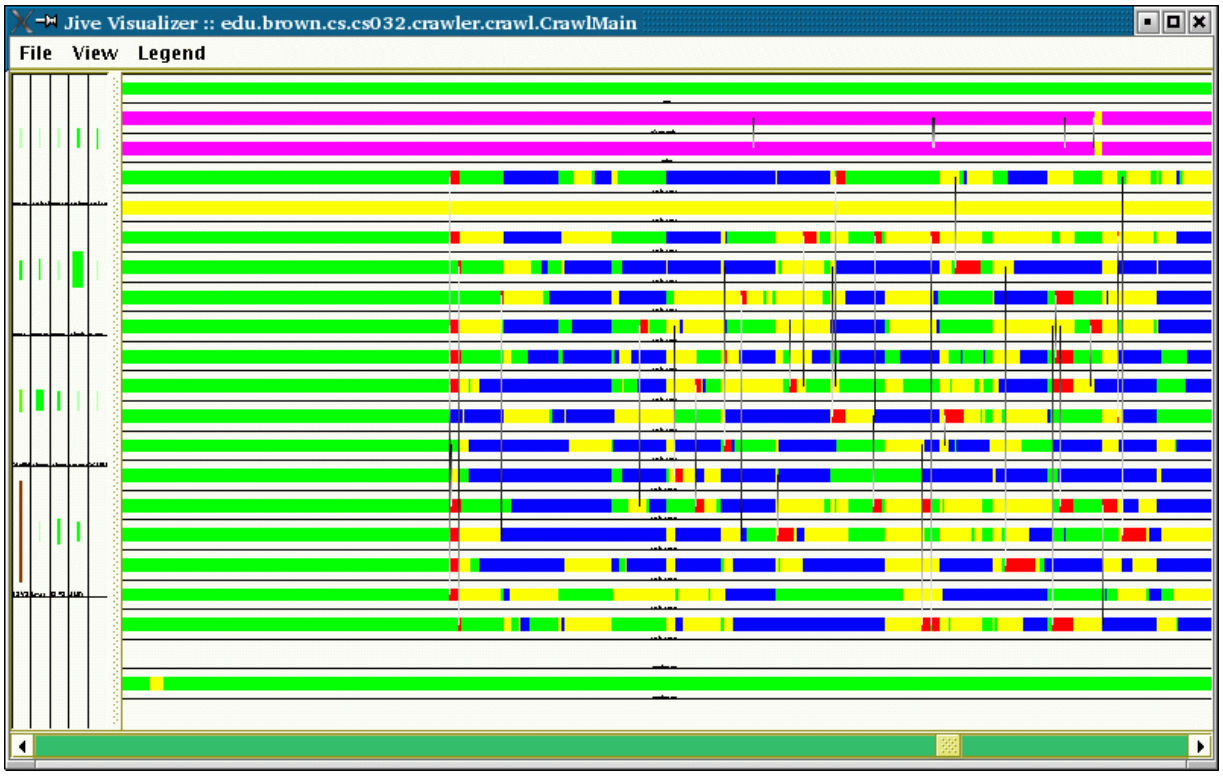

FIGURE 11. JIVE visualization showing thread states along a time line.

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 [47].

The key to making JIVE practical was to ensure that it could be used with real programs in real time. This was done by using fast and minimal instrumentation, summarizing the data, and then sending the summary to the visualization process. Over a wide variety of different programs, the worst performance we have seen with

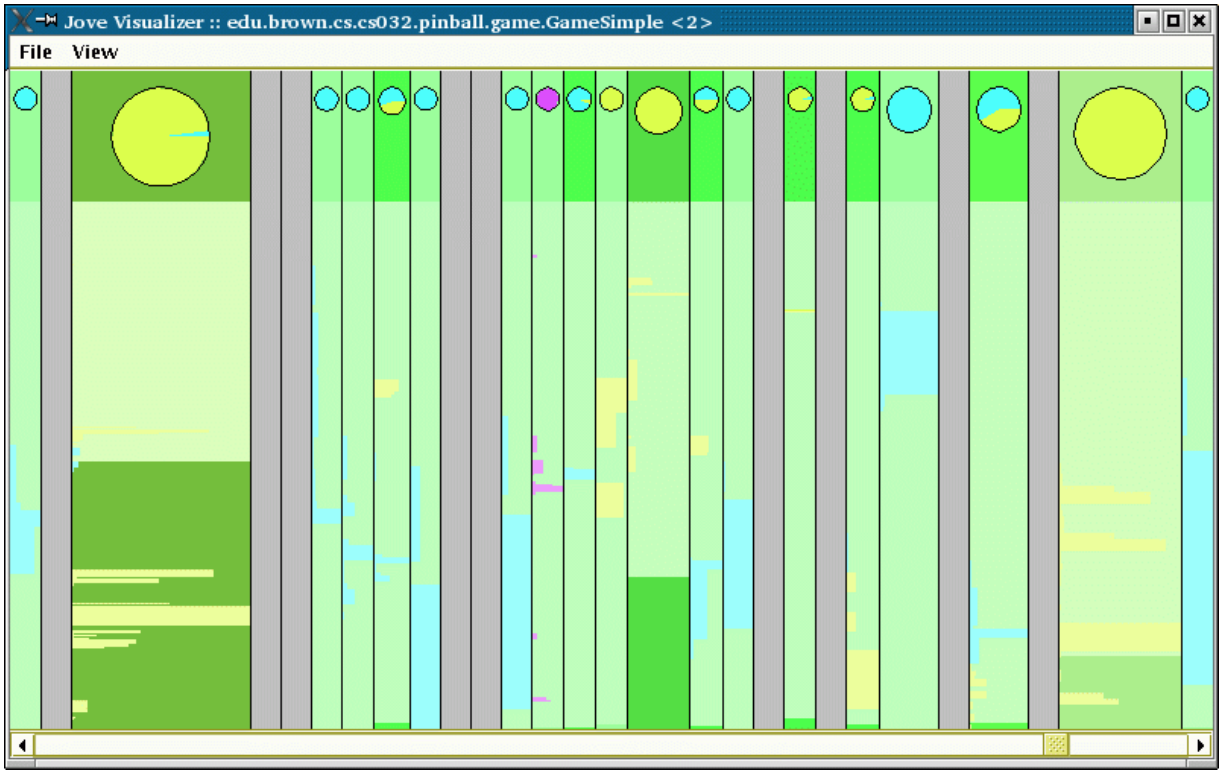

FIGURE 12. JOVE display showing thread usage at the basic block level.

JIVE is a slowdown factor of two, with most programs running at about their normal speed.

4.2 JOVE

A second on-line visualizer, JOVE, maintains a more complex model of program behavior but limits itself to the user's code and not the underlying libraries [48]. 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 executed 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 12. Here each vertical region represents a class. The circle at the top of each bar is used to show how much time each thread spent in that particular class during the interval, using the colors associated with the various threads to

form a simple pie chart. 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 indicates the thread or threads executing those blocks; the width of the line indicates the number of times the block was executed. Again, with JOVE we emphasized minimal, fast instrumentation. JOVE slows the program down by at most a factor of four, and generally a lot less.

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 the HTML parser in J2SE 1.4.

Several other visualization techniques along the similar lines have been used over time. Prominent among these are visualizations that show performance hot spots in the program. This has been done in terms of abstract source files using See-Soft [13], in terms of UML-style interaction diagrams as in Jinsight [33-35], and as a graph of memory (or object) versus time in the hot spot visualizations provided by Evolve [27,55], although these were all done off-line rather than on-line.

These views however suffer much of the same limitations 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 dynamically visualizing program executions is to provide information that is meaningful for understanding the specific but not yet defined problems that programmers actually need to have addressed 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 class and 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.

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. They will have to be dynamically updated as the program runs. They will have to provide enough information to drive appropriate visualizations.

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, waiting to process a robots.txt file, or waiting to write information to disk. Essentially, the programmer has a model of thread behavior for the particular application and wants to see a visual display in terms of that particular 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 (e.g., 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, understand, and implement and they must be reusable. The set of visualizations provided must be flexible and easily adaptable to the different models.

There are several challenges to achieving such visualizations. The first involves finding the right framework for defining the models. This framework must be powerful enough to encompass all the above examples and as well as any others user's might come up with. At the same time, it must be simple enough so that programmers can understand how it can and should be used. One possibility is to use automata over parameterized program events combined with suitable data structures. This combination can model the described situations with minimal overhead.

A second challenge involves defining an appropriately broad set of base visualizations. Multiple visualizations have been defined in several systems including Escalante [28], Cacti [45], and Evolve [55]. Each of these system contains a framework that handles the common elements among the visualizations and is designed so that new visualizations are relatively easy to code and incorporate into the framework.

Customization of the visualization should be up to the user with an appropriate end-user environment.

A third challenge involves providing support to automate or simplify associating the model with the visualization. Many of today's tools do this through the use of wizards and one could probably design wizards to handle many of the more common visualization tasks. Appropriate associations might have to go beyond simple mappings and consider more detailed (but still on-line) analyses. Here one could imagine on-line versions of what has typically been done off-line. For example Walker and Murphy view traces in terms of high-level models based on name patterns [54], Grundy et al. [19] and Sefika et al. [51] abstract the trace using the underlying architectural structure of the system, Pacione provides a variety of abstractions based on program structure and UML diagrams [32], Bertuli et al. abstract run time information using the class structure [3], and De Pauw et al. do pattern analysis of the events as a basis for an compact abstract display [34].

A fourth challenge involves ensuring that the data needed by the visualization models can be derived from program execution with small overhead. While JIVE and JOVE show that we can achieve this for specific data, the challenge involves handling general data and determining exactly what data is relevant to a particular visualization. Systems such as Aspect/J show that program events can be detected efficiently [23]. Jinsight has demonstrated the feasibility of tracing small sections of the program based on trigger events, a technique which can greatly reduce the trace overhead [36]. Dtrace uses application-specific dynamic, low-level probes to minimize instrumentation and its effect and to provide high performance [10].

A final challenge involves putting this all together in a single visual framework and integrating it with a programming environment. This is more of an engineering task that involves developing appropriate user interfaces and making the relatively complex models and visualizations accessible to the programmer. This should build on the vast literature and experience that has been developed for end-user programming and can be integrated into an environment such as Eclipse relatively easily [16].

We can then imagine a new visualization framework. The programmer would be working with their program and run into some problem that requires understanding program behavior, say for example they wanted to understand the internal settings of a finite state automata that their system was modeling. They should be able to quickly specify what they are interested in by defining the relevant program events and state (creation of the automaton, changing the current state), and defining what they want to view (the current state). The system would then instrument the program and provide the corresponding visualization.

6. A Taxonomy of Execution Visualizations

The history of execution visualization can best be summarized by a taxonomy that shows the different properties of previous systems and compares that with what will be needed to create a truly practical and useful visualizer for large, modern systems.

The previous discussion brought out several different dimensions along which the various systems can be compared. The ones that seem the most relevant to under-

standing the practicality of using the visualization for real problems on real systems include:

- *Execution Effect*. This is a measure of how much overhead the visualization puts on the application. As we have noted, some of the visualizations, principally the early ones, introduced so much overhead that the application was unusable. At the other end of the spectrum, visualizations such as the heap viewer or JIVE let the underlying application run in near real time. We specify this as high (for high overhead) indicating 2 orders of magnitude slowdown or more, medium indicating one order of magnitude slowdown, or low.
- *Level of Detail*. The various visualizations vary considerably in how much detail they provide. Some provide information in terms of source lines; some only look at methods; others look at threads; still others look at the overall performance. We specify this by identifying the lowest level for which details are displayed.
- *Degree of Specificity.* The visualizations also vary in whether they provide general information about the system's execution or if they provide information about only a specific portion of the execution. We consider those views that provide information about the control flow to be general, while those that concentrate on a specific aspect, such as memory allocation, are specific.
- *Abstract versus Concrete*. This is a measure of how much abstraction there is from the actual program to the visualization. This can be either concrete, semi-abstract, or abstract to follow the prior discussion.
- *Customization*. This is a measure of how much the viewer can adapt the visualization to their particular needs, either by prespecifying what can be visualized or by controlling the visualization parameters. This dimension's ranges can be either none, user-programmable (with considerable effort required), modifiable display, or easily programmable.

System	Execution Effect	Level of Detail	Degree of Specificity	Abstract vs. Concrete	Customization	Effort Required
PECAN	High	Statement	General	Concrete	None	None
ZStep 95	Unknown	Statement	General	Concrete	None	None
BALSA	Medium	Statement	General and Specific	Concrete	Programmable	Significant
GARDEN	Medium	Construct	General	Concrete	Modifiable	Some
Show & Tell	Unknown	Construct	General	Concrete	None	None
Pict	Unknown	Construct	General	Concrete	None	None
FIELD Editor	Medium	Line	General	Concrete	None	None
FIELD Data Structure Display	High	Field	Specific	Concrete	Programmable	None through Significant
Call graph	Medium	Function	General	Semi-Abstract	Modifiable	None
Class Browser	Medium	Function	General	Semi-Abstract	Modifiable	None
Performance	Low	System	Specific	Semi-Abstract	Modifiable	None
PV	Low	OS details	General	Semi-Abstract	None	None
XMPI	Low	Messages	Specific	Semi-Abstract	None	None
Sun Studio Perf	Low	Routine	General	Semi-Abstract	None	None
I/O Viewer	Low	File Operation	Specific	Semi-Abstract	None	None
Heap Viewer	Low	Memory Operation	Specific	Semi-Abstract	Modifiable	None
Transparent Prolog	Unknown	Clause	General	Concrete	None	None
JIVE	Low	Function or Thread	General	Abstract	Modifiable	None
JOVE	Low	Basic block	General	Abstract	Modifiable	None
New System	Low	Abstraction	Specific	Abstract	Easily Customizable	Some

FIGURE 13. Taxonomy of the various on-line program execution visualizations.

- *Effort Required*. Many of the visualization we have been discussing are generated automatically, but some require programmer intervention or actual code in order to create the display. This dimension reflects the degree of coding required and its range includes none, some, and significant.

The various systems that we have discussed can be categorized along these dimensions as shown in Figure 13. The bottom line of the figure shows the categorization of the framework envisioned in the prior section that would provide specific visualizations designed to handle specific problems.

7. 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 show detailed information about some particular aspect of the execution of 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. The development and implementation of the proposed model 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.

8. References

1. Ronald Baecker, “Sorting out sorting,” 16mm color sound film (1981).

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

3. Roland Bertuli, Stephane Ducasse, and Michele Lanza, “Run-time information visualization for understanding object-oriented systems,” *Workshop on Object-Oriented Reengineering*, (2003).

4. Alan Borning, *Thinglab -- a constraint oriented simulation laboratory*, Ph.D. Dissertation, Department of Computer Science, Stanford University (1979).

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

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

7. Margaret Burnett, Bing Ren, Andrew Ko, Curtis Cook, and Gregg Rothermel, "Visually testing recursive programs in spreadsheet languages," *IEEE Symp. on Human-Centric Computing Languages and Environments*, (September 2001).

8. Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, and Chris Wallace, "End-user software engineering with assertions in the spreadsheet paradigm," *ICSE 2003*, (May 2003).

9. Greg Burns, Raja Daoud, and James Vaigl, "LAM: An open cluster environment for MPI," *Proc. Supercomputing Symposium*, pp. 379-386 (1994).

10. Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal, "Dynamic instrumentation of production systems," *USENIX '04*, (June 2004).

11. Anthony Chan, William Gropp, and Ewing Lusk, "User's guide for MPE: Extensions for MPI programs," *Argonne National Laboratory Report MCS-TM- ANL-98*, (1998).

12. John Domingue, "Visualizing knowledge based systems," in *Software Visualization*: *Programming as a Multimedia Experience*, ed. Blaine Price,MIT Press (1997).

13. Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr., "Seesoft - a tool for visualizing software," AT&T Bell Laboratories (1991).

14. Marc Eisenstadt and Mike Brayshaw, "The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming.," *Journal of Logic Programming* Vol. **5**(4) pp. 277-342 (1988).

15. Marc Eisenstadt and Mike Brayshaw, "The truth about Prolog execution," in *Software Visualization*: *Programming as a Multimedia Experience*, ed. Blaine Price,MIT Press (1997).

16. Erich Gamma and Kent Beck, *Contributing to Eclipse*: *Principles*, *Patterns*, *and Plug-ins*, Addison-Wesley (2004).

17. Ephraim P. Glinert and Steven L. Tanimoto, "Pict: an interactive graphical programming environment," *IEEE Computer* Vol. **17**(11) pp. 7-25 (November 1984).

18. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick, "gprof: A call graph execution profiler," *SIGPLAN Notices* Vol. **17**(6) pp. 120-126 (June 1982).

19. John Grundy and John Hosking, "High-level static and dynamic visualization of software architectures," *Proc. 2000 IEEE Symp. on Visual Languages*, (2000).

20. Peter Henderson and Mark Weiser, "Continuous execution: the VisiProg environment," *ICSE 1985*, pp. 68-74 (August 1985).

21. Robert J. K. Jacob, "A state transition diagram language for visual programming," *IEEE Computer* Vol. **18**(8) pp. 51-59 (August 1985).

22. Raghu R. Karinthi, Mark Weiser, and Incremental re-execution of programs, *ACM SIGPLAN Notices* Vol. **22**(7) pp. 38-44 (July 1987).

23. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, "An Overview of AspectJ," in *European Conference on Object-Oriented Programming*, (2001).

24. 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. Blaine A. Price,MIT Press (1998).

25. Takayuki Dan Kimura, Julle W. Choi, and Jane M. Mack, "A visual language for keyboardless programming," *Washington University CS Tech Report WUCS- 86-6*, (June 1986).

26. Henry Lieberman and Christopher Fry, "ZStep 95: a reversible, animated source code stepper," in *Software Visualization*: *Programming as a Multimedia Experience*, ed. Blaine Price,MIT Press (1997).

27. Babak Mahdavi and Karel Driesen, "Heap hot spot visualization in Java," *McGill U. CS Tech report SOCS-01.8*, (May 2001).

28. Jeffrey D. McWhirter and Gary J. Nutt, "Escalante: an environment for the rapid construction of visual language applications," U. Colorado at Boulder report CU-CS-692-93 (December 1993).

29. Sun Microsystems, "Performance Analyzier, Sun Studio 11," *Sun Microsystems Manual*, (November 2005).

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

31. Brad A. Myers, "Incense: a system for displaying data structures," *Computer Graphics* Vol. **17**(3) pp. 115-125 (July 1983).

32. Michael J. Pacione, "A review and evaluation of dynamic visualization tools," *U. Strathclyde Technical Report EFoCS-50-2003*, (June 2003).

33. 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. Blaine A. Price,MIT Press (1998).

34. Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman, "Execution patterns in object-oriented visualization," *Proc. Conf. On Object-Oriented Technologies and Systems*, pp. 219-234 (1998).

35. Wim De Pauw and Gary Sevitsky, "Visualizing reference patterns for solving memory leaks in Java," in *Proceedings of the ECOOP '99 European Conference on Object-oriented Programming*, (1999).

36. Wim De Pauw, Nick Mitchell, Martin Robillard, Gary Sevitsky, and Harini Srinivasan, "Drive-by analysis of running programs," *Proc. ICSE Workshop of Software Visualization*, (May 2001).

37. Blaine A. Price, Ian S. Small, and Ronald M. Baecker, "A taxonomy of software visualization," *Journal of Visual Languages* Vol. **4**(3) pp. 211-266 (Dec. 1993).

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

39. 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).

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

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

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

43. Steven P. Reiss and Scott Meyers, “FIELD support for C++,” *Proc. USENIX C++ Conference*, pp. 293-300 (April 1990).

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

45. Steven P. Reiss, “Cacti: a front end for program visualization,” *IEEE Symp. on Information Visualization*, pp. 46-50 (October 1997).

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

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

48. Steven P. Reiss and Manos Renieris, “JOVE: Java as it happens,” *Proc. SoftVis ’05*, pp. 115-124 (May 2005).

49. Joseph Ruthruff, Eugene Creswick, Magaret Burnett, Curtis Cook, Shreenivasarao. Prabhakararao, Marc Fisher II, and Martin Main, “End-user software visualizations for fault localization,” *ACM Symp on Software Visualization*, pp. 123-132 (June 2003).

50. David Saff and Michael D. Ernst, “An experimental evaluation of continuous testing during development,” *Proc. 2004 ISSTA*, pp. 76-85 (2004).

51. Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell, “Architecture-oriented visualiazation,” *OOPLSA ’96*, pp. 389-405 (1996).

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

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

54. Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak, “Visualizing dynamic software systems information through high-level models,” *OOPSLA ’98*, pp. 271-283 (1998).

55. Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour, Laurie Hendren, and Clark Verbrugge, “EVolve: an open extensible software visualization framework,” *Proc of SoftVis 2003*, (June 2003).