Issues in Visualization for the Comprehension of Parallel Programs

Eileen Kraemer and John T. Stasko

College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280
E–mail: {stasko,eileen}@cc.gatech.edu

Abstract

Parallel and distributed computers are becoming more widely used. Thus, the comprehension of parallel programs is increasingly important. Understanding parallel programs is more challenging than understanding serial programs because of the issues of concurrency, scale, communications, shared resources, and shared state. In this article, we argue that the use of visualizations and animations of programs can be an invaluable asset to program comprehension. We present example problems and visualizations, showing how graphical displays can assist program understanding. We also describe the Animation Choreographer, *a tool that helps programmers better comprehend the temporal characteristics of their programs.*

1 Introduction

The comprehension of a program or its design is an important component of the coding, debugging, maintaining, testing, and reuse of software. Maintenance, in particular, is an expensive and difficult process. This difficulty is exacerbated by the fact that the maintainer is generally not the author, and that written documentation may be out of date, incomplete, or nonexistent.

A number of models, techniques, program representations, and tools have been developed to address the problems of program comprehension. In the top-down or problem-driven model[5], program comprehension is achieved through the formation, confirmation, rejection, and refinement of expectations of domain concepts. The user may search for confirming *beacons* [1] of these expectations. This method is thought to be common among expert programmers or if the code or type of code is familiar. In the bottom-up or code-driven model[23], comprehension is achieved through the detection of patterns or plans in the program code. This approach is common when the code is completely new to the programmer. In practice, comprehension may proceed top-down, bottom-up or a combination of the two.

All tools for program comprehension must perform three tasks: *extraction* or data collection, *abstraction* or analysis, and *presentation* or display of the result of the analysis. Various tools employ different methods for extraction, abstraction, and presentation, and place varying emphasis on the importance of each of these tasks. The result produced by these tools varies widely, from written documents, through intermediate forms passed on to CASE tools or code generators, to databases that may be browsed or queried, to static and/or dynamic visualizations of the program. Some of these tools produce predefined sets of reports or displays or furnish answers to fixed sets of queries, others allow the analyst to interactively select the analysis to be performed or the information to be displayed.

For example, TIBER(described in [22]) and Synchronized Refinement[21] focus on producing improved documentation. Charon[22] produces an intermediate result that can be passed on to a CASE tool. Ghinsu[15] allows the analyst to interactively explore the program, select target statements and variables, and perform analysis such as slicing, dicing, ripple analysis, and dependence analysis. DOCKET[12] creates a system model that may be interactively queried or browsed by the analyst.

A number of graphical representations of program information have been proposed. These graphical representations may be used internally to calculate results such as program slices or data flow information, or they may serve as the basis for displays that are presented to the analyst. See [7], [17], [8], and [9] for a more detailed discussion of program graph types. Use of these combined program representations allows a program comprehension tool to reference a single in-

[1]This research has been supported in part by a grant from the National Science Foundation (CCR9121607) and a graduate fellowship from the Intel Corporation.

ternal representation, yet construct a variety of views, thus allowing the analyst to gain a better understanding of the program.

Systems such as CARE[14], VAPS[4], and VIFOR[20] visualize program dependencies. VAPS displays include a control flow graph, declaration and nesting trees, and a structure chart. VIFOR uses a two-column entity-relationship display. CARE uses a similar model, but presents a multi-column, *colonnade* display in addition to the traditional call-graph display. PUNDIT[16] combines statically collected semantic information with debugging capabilities, and provides both graphical and textual displays. Graphical displays include a dynamic call graph, an animated control flow graph, and data structure displays.

Other types of *software visualization*[26, 19] systems including BALSA[2], Zeus[3], and TANGO[24] focus on algorithm animation; they support the user in the design of arbitrary visualizations. The graphical displays described in preceding paragraphs, and those constructed using other visualization tools, can be much more intuitive and effective than textual representations, and thus can aid the analyst in assimilating the information produced by the program comprehension tool. These displays are most useful when they closely match the mental model[18] that the programmer/analyst forms through the process of program comprehension. The presentation of multiple views, and a facility that allows the viewer to hide, rearrange, or interact with objects in the display, all assist the analyst in achieving this match, and thus facilitate the comprehension of the program.

Thus, the role of visualization in program comprehension is that of *facilitator*. Visualization can help the analyst develop *intuition* about the functioning of the program under study. This intuition can help the "top-down" analyst form reasonable expectations about the program. The underlying program comprehension analysis tools can extract the information and perform some analysis or abstraction. Visualization can then present this information in such a way that the analyst will easily recognize its importance with regard to the expectation.

Similarly, visualization can help the "bottom-up" analyst proceed in an efficient manner. For example, by viewing an animated call graph of the program under study, the analyst can observe the order and frequency of execution of the various subroutines, and use this information to decide the order in which to study these pieces of code.

In the remainder of this paper, we will focus on a relatively new and extremely challenging program comprehension problem: the comprehension of *parallel and distributed* programs. We will discuss the role that visualization can play in facilitating their understanding. We will point out some of the issues that arise in the comprehension and visualization of parallel programs, and present our approach to addressing these issues.

2 Issues in the comprehension of parallel programs

The introduction of parallelism adds an additional twist to the model of program comprehension. Parallel programs are by nature large and complex. They often produce vast quantities of data. Programmers must understand and analyze large amounts of information describing complex relationships, including the states of each process and interactions among processes. Interactions include communication, synchronization, access to shared variables, and competition for shared resources. In addition to the control and data dependences, and control flow and data flow information that is essential to the understanding of a serial program, the user must address the added complexity of concurrency. The analyst now has additional questions to answer: Which pieces of code may execute concurrently? What type of synchronization must occur between these concurrent threads or processes? Are there any race conditions? Where and how is data shared? The analyst must decipher not only the portions of the code directly related to the purpose of the program, but also must wade through the code that creates and synchronizes processes, allocates tasks, makes intermediate results available to other processes, serializes access to shared memory, and determines some type of global state such as a termination condition.

Furthermore, performance is the primary motivating factor for the creation of parallel programs. Design decisions often hinge on obtaining the best possible performance on a particular architecture, rather than on producing a straightforward implementation of the underlying algorithm. Subsequent ports to other machines and additional "tweaking" (adjustment) can further obscure the original design.

We believe that visualization can assist the user in grasping the concurrency of the program, in managing the large number of objects, in understanding the interactions, and in analyzing the data describing the program's execution. We know that two-dimensional displays of information, such as bar charts and graphs,

give viewers insight into the data presented[28, 29]. Graphical animation can provide additional insight and allow the viewer to absorb more information by tapping into our well-developed visual abilities for detecting patterns, for tracking moving objects, and for spotting anomalies in patterns.

Visualization is a rich medium for communicating information about a program and its execution. That is, it allows program attributes to be represented by colors, shapes, sizes, locations, and motion, rather than merely as a text label and numerical data. When done well, visualization can improve understanding, and make obvious details that would have been obscure in a purely textual presentation. These "information-dense" displays can convey much more information than strictly necessary to confirm or reject a hypothesis. They can supply these answers *and* provide intuition about *why* the hypothesis is false or suggest additional refinement to the viewer.

Why is visualization superior to text for representing many aspects of parallel programs? Textual presentations are inherently serial. As stated earlier, displays are most useful when they closely match the viewer's mental model of the computation. A serial, textual presentation of program information is difficult enough to follow for the programmer who wrote the code and who thus should have a fairly well-established mental model of the concurrency in the program, and who has only to map the text back to this model. The reverse process, required of the analyst who must comprehend unfamiliar code, the *construction* of a mental model of this concurrent process from purely textual, serial data is surely more difficult. An animated, graphical display can more easily convey the concurrency of the program, and can more naturally deal with the temporal issues of parallel programs.

For example, suppose we are dealing with a program containing barrier synchronizations. The "master" process must check in to the barrier before any "slave" process may proceed past its "barrier check-in" statement. All "slave" processes must check out of the barrier before the "master" process may proceed past its "barrier check-out" statement. In the visualization shown in Figure 1, a new two row grid is displayed each time a barrier synchronization is invoked. As each participating process checks in to the barrier, the appropriate circle in the top row is filled in. If the process must wait, its color fades to indicate that it is inactive. When the master checks in, the processes are shown as active again. Similarly, as each participating process checks out of the barrier, the appropriate circle in the second row is filled in. If the master has to wait, it is shown going inactive. When the final slave process checks in, the master process is shown as active again.

Suppose, in our example, that the analyst knows that a barrier synchronization is in use. Do all processes participate in the barrier? If not all, then which ones? Which processes have to wait? Which processes keep the others waiting? An analyst might have to wade through a good deal of text or make a number of queries to answer this question. However, a quick glance at a display such as that shown in Figure 1 can answer these questions.

Similarly, an analyst attempting to understand the pattern of access to a shared variable could easily observe this from the display in Figure 2. In this display, the large circle represents a "mutex" - a critical section of code protected by a mutual exclusion variable. When a process obtains control of the mutex variable, its icon(a colored circle) is shown entering the circle. Processes that are waiting to gain access to that mutex are shown waiting outside the circle. Is there contention for this variable? That is, are there many icons waiting around the circle? How much time does a process spend inside the mutex relative to the time it spent waiting? This type of display would be useful in understanding design decisions such as the use of a distributed list rather than a centralized list. The analyst who must port this code to a new architecture can then make a more informed decision about whether to keep the distributed list or go to a centralized list in the new architecture.

Of course, this same information could be provided without visualization by an "ideal" program comprehension tool - a tool that could answer questions like "Why is there a barrier synchronization at this point?" or "If I add a routine that accesses data structure X, do I now need to enforce mutual exclusion on X?" In such a world, visualization would not be necessary. However, until the time that program comprehension tools have advanced to this state, visualization can serve the useful purpose of providing a rich medium for conveying information - information regarding data and control dependences and flow as in the serial world, and information regarding concurrency, distributed state, and shared variables in the parallel world.

Animated displays, in particular, are useful for conveying information regarding concurrency. They employ the very natural mapping of time to time, rather than the less natural time to space mapping, or the more obscure time to 12-digit timestamp value of a textual report. Events that were concurrent in the

program can be shown as concurrent in the display. In fact, events that *might have been* concurrent can be shown concurrently in the display. This leads us to a discussion of the importance of time and event order in the visualization of parallel programs.

3 Time and event order in the visualization of parallel programs

A number of systems providing visualizations of concurrent programs have been developed[10]. However, not all visualization systems are designed to deal with concurrency. Many follow a serial paradigm in which the visualization system receives some data from the executing program, animates the display to represent that event, and then processes another event (piece of data). Unfortunately, we then lose the concurrency inherent in the program.

The POLKA[25] animation system that we use to develop visualizations can support concurrent animations actions, however. POLKA allows designers to create graphical objects such as lines, text, circles, rectangles, etc., and then make the objects move, resize, change color, flash, and so on. A particular object can be performing many different actions at once, or multiple objects can be changing at the same time, thus reflecting the concurrency of a parallel program. POLKA is implemented on top of the X Window System and Motif.

Concurrent programs often consist of logical phases or rounds. The execution of these rounds or phases may be skewed in time across processors with different workloads or speeds. If valid timestamps are available, the user may wish to view the computation with the events ordered as they actually occurred. However, the calculation of a complete ordering on program events may not be possible - clocks may run at different rates, may have insufficient resolution, or may not be synchronized across processors. Techniques exist to minimize this problem, but they do not eliminate it and they often incur substantial overhead. Synchronization events - message sends and receives, barrier synchronization, serialized access to shared variables - can be used to calculate a partial ordering of events. Within the constraints of partial ordering, a number of feasible orderings exist. Visualizations that adhere to different feasible orderings can give the viewer different perspectives on the computation.

Several authors[27], [13], and [6], have emphasized the value of displaying alternate orderings of a program's execution. To truly comprehend a parallel program, the analyst must understand what these various orderings are and how they can affect the program under study. Of course, the many combinations of possible event orders makes it unmanageable to generate and display *every* feasible ordering. Instead, we utilize an *Animation Choreographer*[11] that provides several useful, canonical orderings, based on the synchronization events produced by the program under study. In addition, the Choreographer allows the user to interactively adjust these to produce any additional orderings.

POLKA and the Animation Choreographer are part of the PARADE (PARallel Animation Development Environment) system for the visualization of concurrent programs. A third component is an instrumentation or monitoring tool. The use of the instrumentation or monitoring tool, which varies between architectures and languages, helps identify the event records for a program.

Using POLKA, libraries of visualizations have been developed - synchronization, history, and callgraph views for Pthreads programs on the KSR(as shown in Figures 1 and 2), 3-D visualizations of communication on the MasPar, algorithmic and performance views of branch and bound algorithms in the iPSC hypercube, as well as a number of application-specific visualizations. Using PARADE, programmers may select visualizations from libraries such as these, or they may create their own new visualizations.

The Animation Choreographer allows the analyst to view program visualizations under a variety of orderings. As an example we will discuss a program that performs a parallel quicksort on an array of numeric values. The parallel execution follows a fork-join paradigm, implementing a tree-structured algorithm. That is, the computation begins with a single processor that reads in the data values to be sorted, and then makes a pass over the data in which it determines a median value and places all elements with values less than or equal to the median value on one side of the array, and all values greater than the median value on the other side of the array. It then forks off two child processes and waits for them to finish their work and join back to it, the parent process. Each of the child processes does the same thing to its portion of the array and forks off child processes of its own. This continues recursively.

The analyst attempting to comprehend the program might want to look at some of the same displays that are helpful with serial programs such as dependence graphs and animated call graphs. In addition, he might like to see application-specific or domain-

specific (in this case, we'll consider sorting as the domain) displays such as those shown in figures 3, 4 and 9 through 11.

Figures 3 and 4 are an animated view of the array. Each bar represents an element in the array (it could also represent a group of elements). The height of the bar indicates the value of the element, and the horizontal position indicates its index in the array. Color is used to show the processor that last touched that element. This use of color allows the viewer to easily detect and follow the actions of a particular processor across multiple displays. At the start of the visualization, the bars are arranged to represent the initial, unsorted array. As elements are swapped in the algorithm the bars are shown changing places in the display. From this display the viewer can observe the order in which elements are swapped, and see which processors act on each portion of the array, giving clues to the functioning of the underlying algorithm.

Figures 9 through 11 are animated swap histories of the parallel quicksort program. Again, horizontal position is used to indicate array index and color is used to indicate the processor performing the swap. Each time two elements are swapped, a horizontal line is drawn between them, in the color associated with the processor that performed the swap. Time runs upward on the display. Thus, in a sequence of swaps, the most recent swap is on top. The triangular patterns in the display indicate that the algorithm works by comparing the processor's first and last elements in its portion of the array, and then working in toward the middle of the subarray. The analyst can also determine the number of processors active at each stage (by counting the number of triangles in a row), and the depth to which the algorithm recurses in this execution (by counting the number of triangles in a column).

The order in which these swap events are displayed can greatly affect the appearance of the displays and the information that can be gained from them. The Animation Choreographer provides four orderings: *Timestamp*, *Adjusted Timestamp*, *Serialized*, and *Maximum_Concurrency*. The Choreographer reads in event records (in this example, each swap of array elements has an associated event record). It then displays an *execution graph*, an acyclic, directed graph in which the nodes represent the recorded program events, and the arcs indicate the temporal precedence relations between these events. The events produced by a particular process or thread are displayed in a column. Arcs between these nodes indicate the sequential relationship between the events of a single process. Arcs between columns are the result of synchronization events such as forks, joins, or barrier synchronizations.

Processors (or threads, etc.) are arranged from left to right. Vertical position in the graph represents execution time, with earlier times appearing above later times. Shape and color of node objects can be used to identify different event types. The execution graph reflects the program events as they were recorded. The user can examine the recorded events by scrolling through the graph, and by clicking on nodes of interest.

To begin viewing a program trace, the user selects an ordering. An initial ordering choice might be a *timestamp* ordering - the execution times are used to order the events for visualization. Figure 8 shows the appearance of the choreographer using events from parallel quicksort and a timestamp ordering. This ordering is frequently very useful to a user wishing to see the actual order of execution, when such information is available. This method relies on the existence of a global clock with adequate resolution, and will produce an essentially sequential visualization under these circumstances. Poor resolution, or timestamps that are not valid across processors, however, may produce visualizations that are misleading or incorrect.

Figure 4 shows the final appearance of the array view when timestamps are used to determine the order of events. As you can see, this does not appear to be a sorted array. In fact, there seem to be "holes" in the array. An examination of event records reveals that there are duplicate timestamps - the clock used was not of adequate resolution. The overlapping event symbols in the Choreographer display of figure 5 are a result of these duplicate timestamps. In a timestamp ordered visualization, all events with the same timestamp are animated concurrently. In this case, swaps that occurred sequentially are animated concurrently because they received the same timestamp. If a given element is involved in multiple swaps with the same timestamp, its final location is not correct, and it may seem to "disappear," resulting in a misleading visualization. Similarly, figure 9 shows the final appearance of the swap history view using a timestamp ordering. There appear to be very few swaps. Again, this is a misleading display resulting from inadequate clock resolution.

Within timestamp ordering we have several choices for scaling. We can use a 1:1 mapping from timestamp units to animation frames. However, this can result in an animation with long periods of inactivity, punctuated by short bursts of activity too rapid for the viewer to comprehend. Another option is to use an n:1 map-

ping from timestamp units to animation frames. This shortens the spans in which nothing happens, but intensifies the short bursts of activity. A third option is to use the timestamps to order the events for visualization, but to ignore them in determining the interevent waiting time. This eliminates the long waits, and allows the event activity to be visualized at a rate the viewer can understand. However, in this type of scaling we lose information about the relative timing of the program events. Ideally, it would be desirable for the mapping from execution time to animation time to behave like a "fun-house mirror." That is, we would like to compress long inter-event times, and stretch out periods of high activity, allowing viewers to discriminate between individual events, but preserving a perspective on the actual timing of events.

The choreographer display under an *adjusted timestamp ordering*, shown in Figure 7, represents a compromise on these goals. In this ordering the timestamps are adjusted just enough so that causal ordering is maintained (the displays are "correct"), but long interevent times are unaffected. This ordering is useful in obtaining a valid visualization without losing the perspective on the true sporadic nature of the program's execution behavior. The use of this ordering results in a "correct" final appearance of the array display, in which the elements are truly sorted, and a "correct" swap history view, in which no element is swapped more than once in any time period.

Figure 6 shows the choreographer under a *serial ordering*. For a serial ordering, we construct a complete ordering of events consistent with the partial order determined by the dependence relations. This method can produce valid, comprehensible visualizations in the absence of globally synchronized timestamps with adequate resolution, such as we have in this example. The array view is again "correct". The swap history view appears as in figure 10. If you look closely you will see that no two swap lines have the same vertical position; the visualization has been completely serialized. We have gained a "correct" ordering, but we have lost the concurrency in the display. In addition, we have eliminated the long interevent times.

Finally, we may wish to use a *maximum concurrency* ordering, as illustrated by the Choreographer display of Figure 8. (For this particular set of events and times, the maximum concurrency and adjusted timestamp orderings are nearly identical - this is not always the case.) In this ordering, we gather all events that *could* have occurred together, and animate them simultaneously. Essentially, this view shows the maximum concurrency possible given the partial order defined by the synchronization events. This ordering also produces a correct array view. The final appearance of the swap history view is shown in figure 11. It is correct, but you will notice that unlike the serial ordering of figure 10 there are multiple swaps on the same line. That is, they are visualized here and in the array view as concurrent actions. We get correctness and concurrency without the long interevent times. In other visualizations we have found this ordering type to be useful for identifying bugs by illuminating concurrent situations that were not imagined by the program's designer.

These various temporal perspectives can provide the user with insight into the program's execution, with each different ordering of the animation shedding light on a different aspect of the computation. The Animation Choreographer allows users to view program animations under the orderings described above, and to specify variations on these orderings.

4 Conclusions

Parallel and distributed programs present a number of challenges to program comprehension. The added complexity introduced by the multiple threads of control, the interactions between processes, the peculiarities of the various parallel programming paradigms and libraries and the tendency to optimize code for a particular architecture exacerbate the problems of program comprehension.

Visualization systems allow program attributes to be represented by colors, shapes, sizes, locations, and motion, rather than merely a text label and numerical data. When done well, visualization can improve understanding, and make obvious details that would have been obscure in a purely textual presentation. We believe that computer visualization, the graphical animation of the behavior and performance of computers and computer programs, can be an effective tool for understanding how programs work.

POLKA is an animation toolkit designed to support concurrent animations. The Animation Choreographer, which relies on POLKA, allows the viewer to easily explore the set of alternate feasible orderings. We believe that this ability to explore the set of possible event orders is essential to the comprehension of parallel programs. We do not claim that visualization will solve all the problems associated with the complexity of parallel programs. However, we do believe that visualization provides a rich means of communicating that can lead to better understanding of how large systems work.

Figure 1: A snapshot of the animated Gthreads barrier display, created by Alex Zhao using POLKA.

Figure 2: A snapshot of the animated Gthreads mutex display, created by Alex Zhao using POLKA.

Figure 3: A snapshot of the array view of the parallel quicksort program under a correct ordering.

Figure 4: A snapshot of the array view of the parallel quicksort program under the timestamp ordering.

Figure 5: A portion of the execution graph produced by the choreographer. The events are from a parallel quicksort program. The ordering type is timestamp.

Figure 6: A portion of the execution graph produced by the choreographer. The events are from a parallel quicksort program. The ordering type is serialized.

Figure 7: A portion of the execution graph produced by the choreographer. The events are from a parallel quicksort program. The ordering type is adjusted timestamp.

Figure 8: A portion of the execution graph produced by the choreographer. The events are from a parallel quicksort program. The ordering type is maximum concurrency.

Figure 9: A snapshot of the swap history view of the parallel quicksort program under the timestamp ordering.

Figure 10: A snapshot of the swap history view of the parallel quicksort program under the serialized ordering.

Figure 11: A snapshot of the swap history view of the parallel quicksort program under the maximum concurrency ordering.

References

[1] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[2] Marc H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, May 1988.

[3] Marc H. Brown. ZEUS: A system for algorithm animation and multi-view editing. In *Proceedings of the IEEE 1991 Workshop on Visual Languages*, pages 4–9, Kobe Japan, October 1991.

[4] G. Canfora, A. Cimitile, and U. DeCarlini. Vaps: Visual aids for pascal software comprehension. In *Proceedings of the Program Comprehension Workshop*, pages 13–15, 1992.

[5] T. A. Corbi. Program understanding : Challenge for the 1990's. *IBM Systems Journal*, 28(2), February 1989.

[6] Janice E. Cuny, Alfred A. Hough, and Joydip Kundu. Logical time in visualizations produced by parallel programs. In *Visualization '92*, Boston, MA, October 1992.

[7] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its

use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3), July 1987.

[8] M. Harrold and B. Malloy. A unified interprocedural program representation for a maintenance environment. In *Proceedings of the Conference on Software Maintenance*, pages 138–147, Sorrento, Italy, October 1991.

[9] David Kinloch and Malcolm Munro. A combined representation for the maintenance of c programs. In *Proceedings of the Program Comprehension Workshop*, pages 119–127, 1993.

[10] Eileen Kraemer and John T. Stasko. The visualization of parallel systems: An overview. *Journal of Parallel and Distributed Computing*, 18(2):105–117, June 1993.

[11] Eileen Kraemer and John T. Stasko. Toward flexible control of the temporal mapping from concurrent program events to animations. *Proceedings Eighth International Parallel Processing Symposium*, pages 902–908, 1994.

[12] P.J. Layzell, R. Champion, and M.J. Freeman. Docket: Program comprehension-in-the-large. In *Proceedings of the Program Comprehension Workshop*, pages 140–148, 1993.

[13] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler. Analyzing parallel program execution using multiple views. *Journal of Parallel and Distributed Computing*, 9(2):203–217, June 1990.

[14] Panagiotos Linos, Philippe Aubet, Laurent Dumas, Yan Helleboid, Patricia Lejeune, and Philippe Tulula. Facilitating the comprehension of c programs: An experimental study. In *Proceedings of the Program Comprehension Workshop*, pages 55–63, 1993.

[15] Panos E. Livadas and Scott D. Alden. A toolset for program understanding. In *Proceedings of the Program Comprehension Workshop*, pages 110–118, 1993.

[16] David P. Olshefski. Position paper: Tools facilitating software comprehension. In *Proceedings of the Program Comprehension Workshop*, pages 32–34, 1992.

[17] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Notices*, 9:177–184, May 1984.

[18] Cherri M. Pancake and Sue Utter. Models for visualization in parallel debuggers. In *Proceedings of Supercomputing '89*, pages 627–636, Reno, NV, November 1989.

[19] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, September 1993.

[20] V. Rajlich, N. Damaskinos, P. Linos, and W. Khorsid. Vifor: A tool for software maintenance. *Software - Practice and Experience*, pages 67–77, January 1990.

[21] Spencer Rugaber. Reverse engineering by simultaneous program analysis and domain synthesis. In *Proceedings of the Program Comprehension Workshop*, pages 45–47, 1992.

[22] Oreste Signore and Mario Loffredo. Charon: a tool for code redocumentation and re-engineering. In *Proceedings of the Program Comprehension Workshop*, pages 169–175, 1993.

[23] E. Soloway and K. Ehrlich. Empirical studies of programming knowledgekk. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984.

[24] John T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

[25] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

[26] John T. Stasko and Charles Patterson. Understanding and characterizing software visualization systems. In *Proceedings of the IEEE 1992 Workshop on Visual Languages*, pages 3–10, Seattle, WA, September 1992.

[27] Janice M. Stone. A graphical representation of concurrent processes. *SIGPLAN Notices*, 24(1):226–235, January 1989. (Proceedings of the Workshop on Parallel and Distributed Debugging, Madison, WI, May 1988).

[28] E. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, CT, 1983.

[29] E. Tufte. *Envisioning Information.* Graphics Press, Cheshire, CT, 1990.