The Automatic Animation of Concurrent Programs

BLAINE A. PRICE, RONALD M. BAECKER

Dunamic Graphics Project, Computer Systems Research Institute, University of Toronto, CANADA, M5S 1A1
Email: r mb@dgp.toronto.edu

Abstract

Much of the program visualization research to date has been devoted to hand-crafted animations of small sequential programs for use in computer science education. Instead, our work focuses on the development of automatic concurrent program visualization tools for use in software engineering. This paper describes a framework for concurrent program animation and a prototype tool based on this framework. Our user testing experiments with the prototype showed a significant increase in programmer insights when compared with conventional tools.

Introduction

Effective communication of complex ideas is one of the most important contributions to human progress. Because scientists and engineers build upon the work of others, it is vital that they present their work in a language that is concise, unambiguous, and expressive. Mathematicians use special symbols to state theorems and proofs. Engineers and architects use diagrams to specify how a structure is to be built. Computer scientists use programming languages to specify what a computer program is to do, but these languages are primarily designed as a communication vehicle from human to machine, not from machine to human.

Program visualization (PV) facilitates the human understanding and effective use of computer programs; it relies on the crafts of typography, graphic design, animation, cinematography, and interactive computer graphics. In effect, PV is an attempt to create an interface that allows one programmer to understand the work of another. Examples include a professor explaining a sorting algorithm to a class and a software engineer working with a team of thousands to produce a million lines of code.

[1]Author's current address: Human Cognition Research Laboratory, Open University, Milton Keynes MK7 4AA, England.

Very little work was done in the field of PV until the 1980's. Haibt developed a system in 1959 that created flowcharts from FORTRAN programs [Haibt 1959]. In 1966, Knowlton created a black and white motion picture that animated list manipulations [Knowlton 1966]. Work in the 1970's led to the production of the colour motion picture *Sorting Out Sorting* [Baecker 1981] which illustrates the operation of nine different sorting algorithms. It is still widely used in high schools and universities as a teaching tool.

The major work of the 1980's was Brown's ACM Dissertation Award-winning thesis [Brown 1988] which describes the BALSA algorithm animation system. BALSA allows programmers to mark "interesting events" in their Pascal programs with calls to animation routines. The user may then run the program using BALSA on a workstation and observe a trace of the program's execution as it runs. Statements are highlighted as they are executed. The user can control the speed of the program as well as run it backwards if desired. A graphical animation is also displayed as the program runs. This animation is driven by the "interesting event" calls inserted by the programmer.

Several workstation-based interactive systems were developed in the latter half of the 1980's, including Pecan [Reiss 1985], the Transparent Prolog Machine [Eisenstadt and Brayshaw 1987], and Amethyst [Myers et al. 1988]. Other systems developed during this period, such as VIPS [Isoda et al. 1987], are advertised as "visual debuggers" or "visual program tracers" and serve much the same purpose as the other PV systems. Unfortunately, these systems are only capable of displaying small "toy" programs and are not appropriate for large scale software engineering.

With the escalating cost of software development, especially on large projects, interest has increased in computer-aided software engineering tools. Fred Brooks claims that there is "no silver bullet" to attack this problem and that PV in particular cannot express the complexity of a large computer program [Brooks 1987]. This view is unrealistically pessimistic since it assumes that software engineering is so difficult a task that a clearer expression of the problem cannot help.

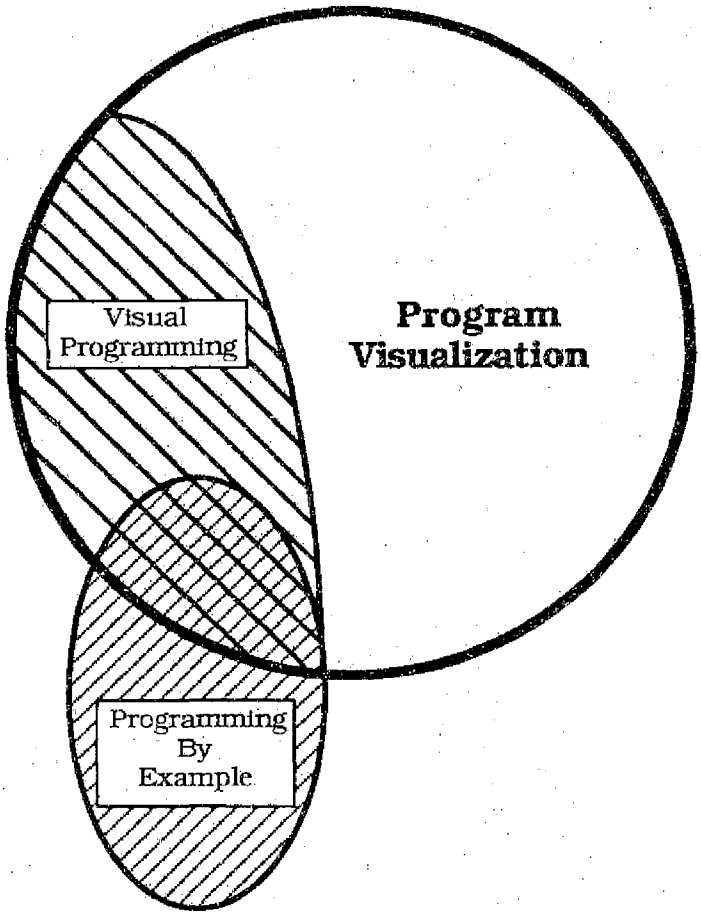

Figure 1: The Relationship Between Program Visualization, Visual Programming, and Programming By Example

Cartographers and chemists have developed notations for expressing very large and complex sets of data from nature in a concise form — why should man-made data be *more* difficult to express? Program Visualization is not a panacea for all of the problems in software engineering, but it does have the potential to increase significantly the communication bandwidth between software engineers and to lead to a better understanding of how large systems work.

A Brief Taxonomy of Program Visualization

Program visualization is often confused with two related areas: visual programming and programming by example. Visual programming is the *specification* of a computer program using graphics[2] while program visualization is the use of graphics to enhance the understanding of a program (that has already been written). Programming by example involves specifying a program by giving (possibly graphical) examples of the input and output data and having the computer infer the program. It is clear that visual programming overlaps with programming by example in cases where the examples are given graphically. If a program is specified graphically, then the specification itself is a graphic image that enhances the understanding of the program in some way, so visual programming is a proper subset of program visualization, as indicated in Figure 1.

One of the first complete taxonomies in this area was that of Myers who divided PV systems along two axes: whether they illustrate the *code* or the *data* of the program and whether the display is *static* or *dynamic* [Myers 1986] (updated as [Myers 1990]). Brown further divided dynamic displays [Brown 1988] into those that are *passive* (such as a motion picture) or *interactive* (such as those requiring user interaction at a workstation). Brown also introduced the term *algorithm animation* to describe dynamic displays that show the fundamental operations of the algorithm in the program, as opposed to *program animation* where the details of the code itself are illustrated.

The taxonomies of Myers and Brown were relevant in the 1980's when the common workstation had a relatively slow (one MIPS) processor with a 1000x1000 pixel single colour display, a simple windowing system supporting a few fonts, with a single "beep" sound available. This was a tremendous improvement over the

[2] By "graphics" we mean displays that use more than single-font, single-colour text.

technology of the 1970's when the common workstation was a VT-100 style terminal which had a 24x80 character display in a single colour (either green or amber) using a single font and a single "beep" sound.

Today, the common workstation is RISC based and has a relatively fast (minimum 25 MIPS) processor with a separate high speed graphics engine and at least 8 bits of colour. Modern windowing systems are network transparent and support a large number of fonts while the hardware supports multi-voice digital waveform sound. It is significant that this technology is highly underutilized by software engineers. In a typical software engineering laboratory, one finds that the processors are virtually idle as programmers use a simple editor in several windows to view parts of their program in a single colour, single font text. The only sound that the editor makes is a simple "beep," even though the workstation can address the full 16,000 Hz of human hearing. The programmer still uses much the same interface as he did twenty years ago, albeit with more real estate.

With the hardware support provided by this new technology, *colour* and *sound* have become important PV categories for increasing the communication bandwidth between machine and programmer. Several authors have suggested how colour [Tufte 1990] and sound [Gaver and Smith 1990] may be used to communicate tremendous amounts of complex real-time information, yet few PV systems take advantage of either.

Although high speed computers have led to great advances in scientific visualization of natural phenomena, software engineers have been resistant to using to using the technology that they helped to create. One reason that PV technology is not employed in software engineering is the perceived cost. The best program and algorithm visualizations have been custom designed and require a great deal of manual code annotation. Since most PV systems only work for small "toy" programs, a software project manager would be unlikely to devote scarce manpower to annotating code or designing visualizations. A good software engineering tool must have a low overhead in relation to the service that it provides. Thus *manual* systems are inappropriate and *automatic* (or nearly automatic) program visualization systems are required.

By "automatic program visualization," we mean a system that is capable of producing a useful default visualization from an arbitrarily large piece of unannotated software with little or no human effort. A good PV system allows the user to select desired views from the default visualization and customize them to his needs. Some authors have argued that automatic visualizations contain too little information to be useful. Yet Baecker and Marcus showed how static typographic techniques could be used to make C programs more readable using a simple parsing procedure [Baecker and Marcus 1990]. Since we now have fast processors which spend most of their time idle, it is possible to apply computationally expensive artificial intelligence techniques to analyze "plans" in source code and determine the underlying algorithm automatically for visualization.

Most PV systems have been designed to deal with *sequential* programs only, even though *concurrent* programs are becoming more popular due to the increasing use of parallel architectures to increase performance. Concurrent programs are among the most complex both to write and debug and are thus likely to benefit from good visualization techniques. It should also be possible for a concurrent PV system to visualize any sequential program (to some extent) since these are a special case of concurrent programs.

Although PV research has been active since the 1970's, it was not until the late 1980's that PV work began to deal with concurrent programs. In 1986, Delisle and Schwartz animated the message passing behaviour of Communicating Sequential Processes [Hoare 1978] using two view windows [Delisle and Schwartz 1986]. Zimmermann et al. animated process behaviour in Portal (a Modula-like language) [Zimmermann et al. 1988] while Socha et al. developed the Voyeur system [Socha et al. 1989] which allows users to manually create views of running concurrent programs using a toolkit. It is important to note that each of these systems lacks a coherent framework from which views are derived: the choice and design of displays appear arbitrary.

One of the problems with the visualization of concurrent programs is the observation effect: adding visualization code can change the relative execution rates of parallel processes so that a visualized program runs differently from a non-visualized program. Although sequential PV systems like BALSA cause code to be added to programs, they are actually *benign* since there is only one thread of control in a sequential program and the addition of code merely slows the whole program down. A visualization would be truly *disruptive* if it caused a concurrent program to synchronize differently than it would have without the visualization. Zimmermann et al. avoided the observation effect by using a special piece of hardware to monitor the program at the bus level so the visualization is benign. Although this effect is important, a correct concurrent program that does not rely on real-time events will always synchronize correctly no matter how much a process is delayed by visualization.

The remainder of this paper describes a framework for automatically animating concurrent programs based on a simple epistemological approach, an implementation based on the ideas in this framework, and an experiment designed to test the utility of our ideas.

A Framework for the Automatic Animation of Concurrent Programs

We base our framework for automatically animating concurrent procedural programs on an epistemological approach: what can one know automatically about a concurrent program? We begin by looking for places where information about the program may be found, the most obvious of which is the *program source code*. Since we know the implementation language, it is easy to parse the program to determine such things as:

- the names and types of data structures
- the names of modules, monitors, and subroutines
- the hierarchical relationships
- the caller-callee relationships
- the data import/export relationships.

All of this information is static and may be used as a "background" upon which other information is animated as the program runs. A second obvious source of information is the *hardware platform* upon which the program runs. Since the hardware configuration does not change for a given visualization, it is possible to know such things as:

- the number of processors (CPUs)
- the amount and type of memory available (shared/non-shared)
- the way in which processors communicate with each other and memory.

There is a third element of information which gives life to the other two in a running program: the *process*. A running process is an executing instance of some position in the code for a particular program on a specific CPU. A running process will cause the program counter in the CPU to change as subsequent instructions are executed, which corresponds to the execution of statements in the high level language of the source code. These instructions might cause the CPU to modify the contents of memory, which would correspond to changes in the contents of data structures. Since processes are the "driving force" in a concurrent program they are often an informative element to visualize.

Our framework has three distinct, concurrent views: the *source-based view*, the *process-based view*, and the *hardware-based view*. For each view, the named element remains static while the other elements animate around it. By keeping one element stationary, one is able to see how all of the other elements interact with the static element: one gets a sense of "what life is like" from its point of view. Each view may also be collapsed to an icon if it is not of interest to the user.

The static version of the source-based view provides a hierarchical call graph diagram of the source code beginning at the highest level of abstraction. The user may manipulate this view and look at deeper levels in the abstraction, including the code itself. This diagram becomes a background for the animation, which appears while the program is running. The animation represents each of the processes and their current location in the diagram.

The process-based view contains a static representation of each process with the software state (running, blocked, or waiting) and hardware state (which CPU it is using) animated during execution. The hardware-based view shows a static diagram of the hardware (CPUs, memory, and devices) with an animated representation of each process indicating which resources a process is using.

A Prototype Implementation

We implemented a prototype of this framework, called "*Paradocs*," for the source-based and process-based views using the language Turing Plus [Holt and Cordy 1988] (which is similar to Ada or Modula III) on a Sun 3/60 using the X Window System [Scheifler and Gettys 1986]. Before compilation, the source code was parsed to obtain the static information for these views. The dynamic information used in the animation was captured by inserting light-weight calls to *Paradocs* in the source code during the parsing phase. A more benign approach using special hardware was unnecessary at the prototype stage because the programs we visualized were unaffected by the changes in the code.

Since the animation is tied directly to program execution, it always indicates the current state of the program. The user adjust the speed and even stop the program (and thus the animation) at any time through the use of a control panel (Figure 2). Our "Replay" feature allows the user to stop the program and watch a section of the animation again at a different speed, either forwards or backwards.

Figure 3 shows the static source-based view for an example program solving the Dining Philosopher's Problem [Dijkstra 1965]. Each rectangular icon with a drop-shadow represents a module. The module with the double border, labelled "Chopsticks," is a monitor [Hoare 1974]. A monitor is a special kind of module which facilitates process synchronization by allowing only one process to execute within it at any time. In the diagram, the module called "Output" has been opened by the user to reveal the data and subroutine that it encapsulates (which may in turn be opened). The user has also selected the *info* button for this module, which revealed a pop-up window explaining the module's function. This information was gathered from the module's leading comments, extracted when the source code was parsed.

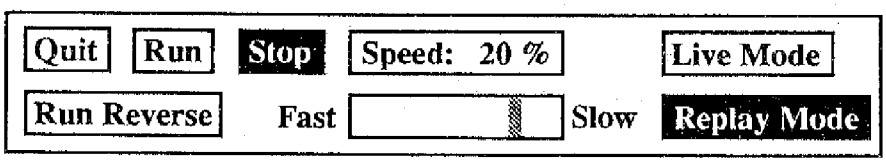

Figure 2: The *Paradocs* Control Panel (program stopped in replay mode)

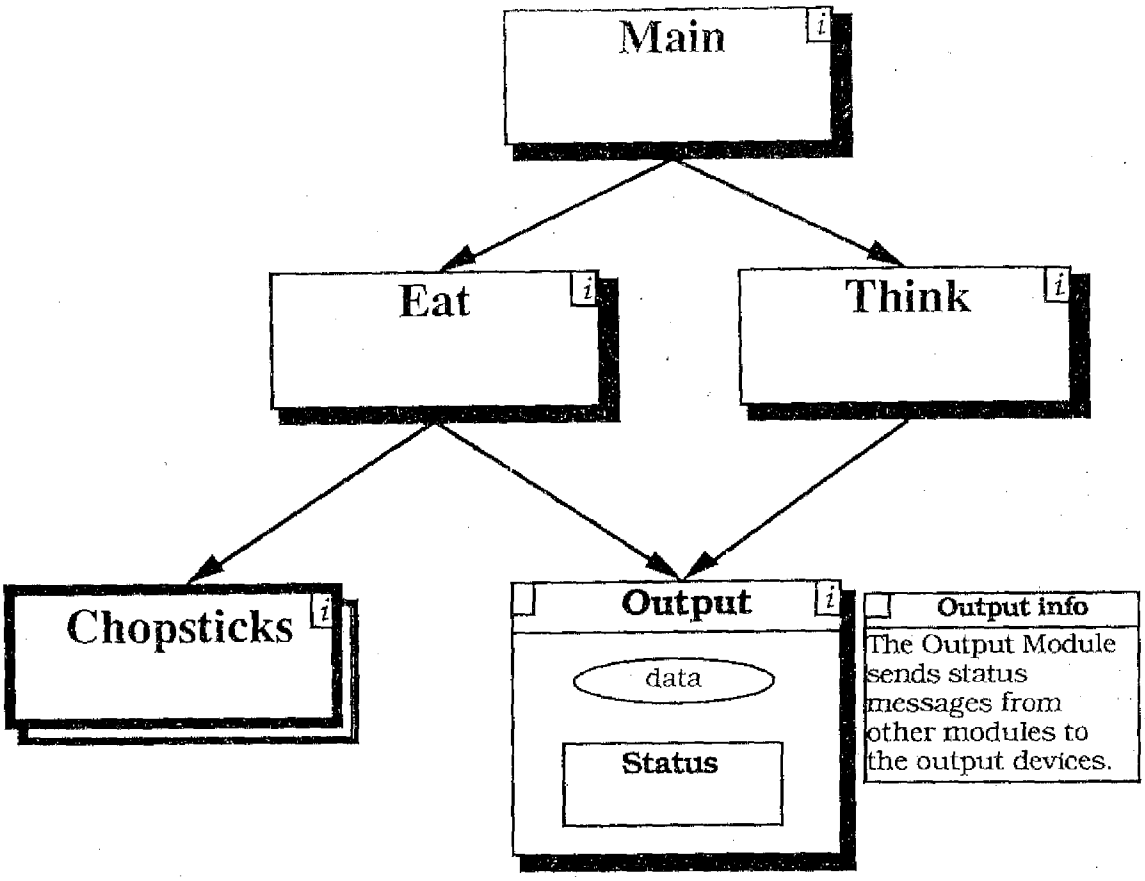

Figure 3: The *Paradocs* Static Source-Based View

When the program is running, each new process appears on the diagram as a numbered, coloured circle. The circle's number corresponds to the process's number and different colours are used to visually differentiate each of the processes. For a given process, the circle appears in the icon representing the module or subroutine where the process is currently executing. If many circles appear inside an icon then a high degree of parallelism is being achieved in that area.

Processes entering a monitor must queue since only one is allowed inside at a time. Figure 4 shows where the circles are positioned depending on their status. The one process currently executing in the monitor appears inside the icon while the processes waiting to enter the monitor for the first time appear on top of the icon at the point of the incoming arrow. If a process running in the monitor is blocked on a condition then it must move outside the monitor to a condition queue and wait to be signalled by

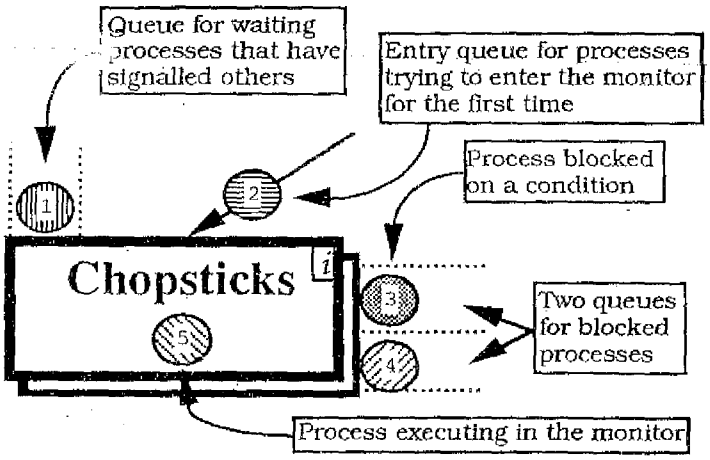

Figure 4: Processes Queuing Around a Monitor

another process. The condition queue appear along the sides of the monitor icon with a separate queue for each condition. When one process signals another that bis blocked, the signalling process immediately leaves the monitor, thus allowing the blocked process to run. The signaller waits temporarily in a queue at the top left corner of the icon.

The source-based view is useful for showing problems with recursion and process synchronization since the state of each process is visible at all times. A process in infinite recursion will appear to continuously re-enter a subroutine, while a poorly synchronized algorithm will have many processes queueing at a monitor and a low degree of parallelism. Table 1 summarizes some ways in which the views in this framework may be used to analyze concurrent program behaviour.

The process-based view uses a notation consistent with the source view: the static representation of a process is a large, numbered, coloured circle. Each large circle in the process-based view corresponds to one of the animated circles in the source view and the same colour is used for quick visual identification. A circle appears when a new process is forked and remains visible for the life of the process. The large process circle may be moved to any position on the screen that the user desires, or it may be collapsed into an icon if it is not of interest. Figure 5 shows the process view positioned for the Dining Philosophers Problem to reflect the circular arrangement of philosophers.

When the program runs, the process-based view indicates the state of each process and its position in the code. The state of each process is expressed by its border, with a solid border showing that the process is running on a CPU, while an idle process will have a discontinuous border. In Figure 5, processes 2 and 4 are using CPU while processes 1, 3, and 5 are not. The thickness of the border indicates the process's software state: processes ready to run have a thin border, medium borders show a process that is waiting, and blocked processes use a thick border. The latter two states do not require a CPU so they are likely to appear as discontinuous lines. Line thickness and continuity were chosen to represent these items because they easily catch the eye and draw attention to blocked and CPU-starved processes.

Activity / Problem	Views		
	Source-Based	Process-Based	Hardware-Based
Deep Recursion	process remains in the same place; upon close inspection it appears to re-enter the same procedure repeatedly	call stack grows very large, but eventually peaks and begins shrinking	high CPU usage
Infinite Recursion	process remains in the same place; upon close inspection it appears to re-enter the same procedure repeatedly	call stack grows without bound	high CPU usage
Infinite Chatter	a group of processes move in a cyclic pattern without making any progress		a cyclic pattern of process/processor communication
Deadlock	processes appear motionless, waiting on conditions outside monitors	all processes are blocked (have thick borders); no process can signal them	CPUs are idle
Starvation	one or more processes remains motionless, blocked on a condition	certain processes never unblock (always have thick border)	certain processes never use CPU
Slow Device Access	process remains motionless in a particular area	processes remain in same state or take a long time to change state	a particular process uses a device excessively
Long Comput-ation	a process remains motionless in an area; close inspection shows that it is simply executing a lot of code	the process continues to use CPU	
Poor Synchron-ization	processes spend a lot of time in a monitor; many processes are trying to get in	lots of processes in wait state (medium thick border)	low CPU usage

Table 1: Activities and Problems Indicated by Views

Figure 6 shows two processes from the Dining Philosophers Program. Process 2 on the left is running on a CPU since it has a thin, solid border. The names of the module and procedure where it is currently executing are shown at the top and bottom of the circle while the left side shows a diamond shaped icon indicating the CPU where the process is running. The icon at the right of the circle shows the current number of levels of subroutine nesting for the process. Process 3 on the right has a thick discontinuous border which quickly identifies it as a blocked process that is not using a CPU. The additional box at the bottom of the circle gives the name of the condition that it is blocked on.

User Testing

A common complaint about PV systems is that they are simply toys and that they are not useful outside the limited domain of novice to intermediate computer science instruction. "Proving" the usefulness of a PV system as a software engineering tool, however, is a difficult task. The scientific method states that the only way to prove a hypothesis is to test it through reproducible experiments, yet few authors have checked their systems with formal user testing experiments. One reason for this is the lack of good experimental methodology in the field of software psychology, which has been described as "an unholy mixture of mathematics, literary criticism, and folklore" [Sheil 1981].

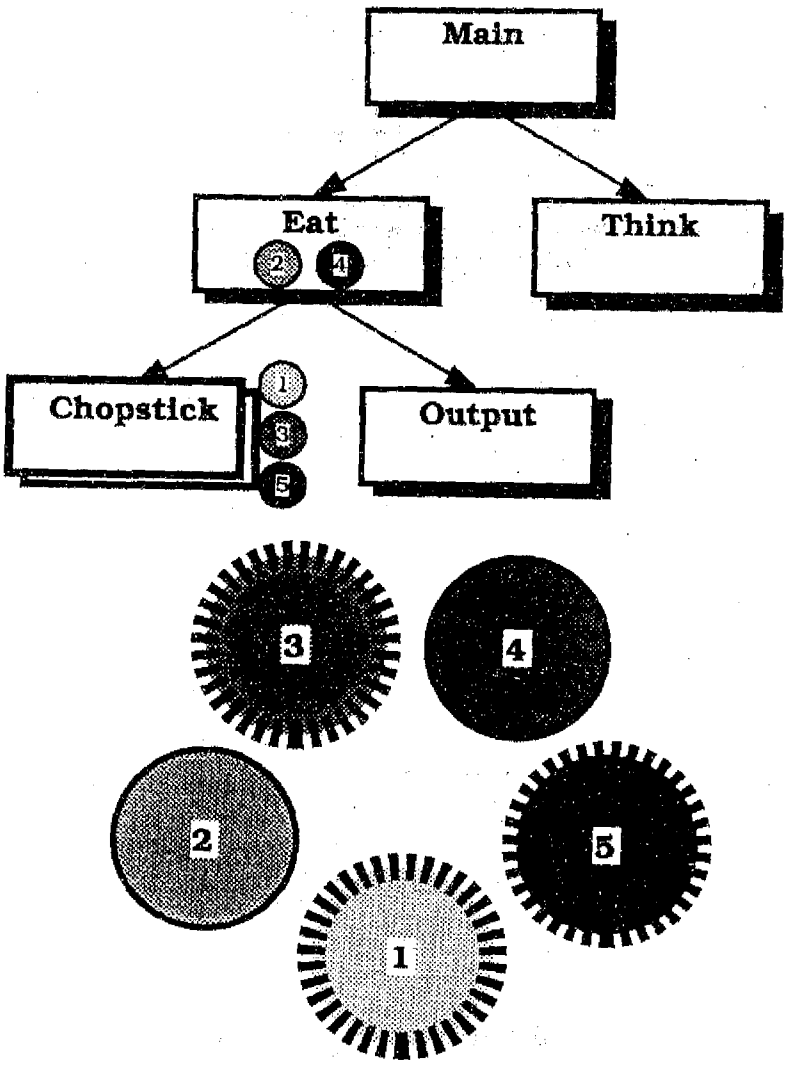

Figure 5: Source-Based and Process-Based Views Positioned By User (Dining Philosophers Problem)

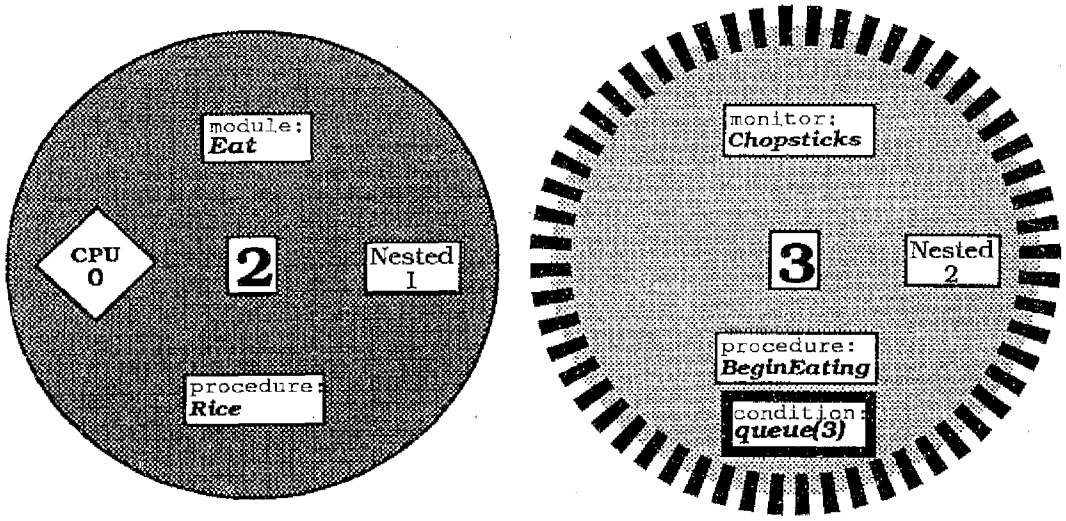

Figure 6: Two Processes in Process-Based View

We performed a user testing experiment using *Paradocs* to determine if it aided in software comprehension for a large modular program. We chose a debugging task to test program comprehension (since one must usually understand a program in order to debug it). Based on experience with pilot subjects and advice from experts, we inserted a bug in a large (7500 lines in 12 modules) operating system simulator called Mini Tunis. We used a between-subjects strategy: one group attempted to find the bug using conventional methods while the second group used *Paradocs*.

The subject pool consisted of graduate and senior undergraduate students taking a computer science course on operating systems. All of the students had been working with Mini Tunis for six weeks while doing course assignments. A total of 20 volunteers from the class were randomly assigned to two groups: the control group (using conventional tools) and the *Paradocs* group. Each group had identical preparation for the experiment, including a familiarization session with *Paradocs*.

Both groups began using conventional debugging tools to solve the problem, but at the fifteen minute mark the *Paradocs* group was allowed to use *Paradocs* to continue debugging. Subjects from either group who did not find the bug after forty-five minutes were stopped and they were recorded as not finding the bug.

The initial fifteen minute period was designed to catch high-ability subjects who could find the bug with or without software aids. Two subjects (one from each group) found the bug before the fifteen minute mark, and their results were not counted further. Of the remaining nine subjects in each group, the raw numerical results for both groups were identical: five subjects found the bug within forty-five minutes and four subjects did not find the bug (the mean and medium times to completion were also identical).

The apparently neutral results only represent a portion of the data, however. All of the sessions were videotaped and the subjects were asked to "think aloud" as they worked. A basic analysis of the videotape revealed that some of the subjects who ran out of time were "close" to finding the bug: those in the control group were examining the routine containing the bug while those in the *Paradocs* group were replaying the animation at the point where the bug was occurring. The verbal protocol from these subjects revealed that they understood the cause of the problem and would likely have found the bug.

The remainder of the subjects who ran out of time were

	Control	*Paradocs*
Solved	5	5
"Close"	1	3
"Lost"	3	1

Table 2: Numerical User Testing Results

clearly "lost" and had little hope of finding the bug. The verbal protocol analysis indicated that they had little idea as to the cause of the bug and the videotape indicated that they were looking in the wrong area of the program. Table 2 shows the numerical results for those who found the bug, were "close," or were "lost."

Further analysis of the videotape revealed that subjects in the *Paradocs* group had more insights into the cause of the bug and made more leaps of understanding than those in the control group. *Paradocs* subjects had a great deal more confidence in their verbal assertions whereas control subjects tended to guess at conjectures without any evidence. *Paradocs* subjects also made extensive use of the "replay" feature to narrow down the location of the bug, as shown by the following excerpt from the transcript of a session:

> `Oh, something interesting here.` *—indicates process being signalled—rewinds animation and replays the sequence again slowly—* `That's not supposed to happen! The init process already signalled another envelope to come in, so the bug is somewhere here...` — *indicates subroutine where bug has been inserted*

Many *Paradocs* subjects also spent a lot of time staring at the animation in an almost mesmerized state. This was probably due to their lack of familiarity with the system and it likely contributed to their debugging time.

Conclusions

We have argued that the capabilities of modern workstation technology far exceed the degree to which they are exploited by program visualization interface designers. We have also asserted the need for *automatic concurrent* program visualization systems as software engineering tools. By building a prototype system based on a systematic framework and performing user testing experiments, we have illustrated that program visualization can benefit from an organized rather than an ad hoc approach. Despite the serious problems with methodology, it is important for PV system designers to scrutinize their work through experiments, even if the results are only qualitative.

Researchers developing concurrent PV systems must be careful to use benign methods in sensitive systems and address the issues of different architectures and paradigms. While scrolling and zooming techniques may work well in simple documents, automatic PV systems must provide tools for effective navigation through the enormous information spaces of large software projects. Despite these research issues, our work suggests that the effective use of graphic design principles, colour, and audio will lead to concise and expressive notations for communicating about complex computer programs.

Acknowledgments

We are indebted to Abigail Sellen for her advice on the design of user testing experiments. We also wish to thank the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of the Province of Ontario, and Apple Computer, Inc. for their support.

References

[Baecker 1981] Baecker, Ronald M. *Sorting Out Sorting.* Dynamic Graphics Project, Computer Systems Research Institute, University of Toronto. 16 mm colour sound film, 25 minutes, presented at ACM SIGGRAPH '81. 1981.

[Baecker and Marcus 1990] Baecker, Ronald M., and Aaron Marcus. *Human Factors and Typography for More Readable Programs.* Reading, MA: Addison-Wesley, 1990.

[Brooks 1987] Brooks, Fred P. "No Silver Bullet: Essence and Accidents of Software Engineering." *IEEE Computer* 20(4): 10-19, 1987.

[Brown 1988] Brown, Marc H. *Algorithm Animation. ACM Distinguished Dissertations.* Cambridge, MA: MIT Press, 1988.

[Brown 1988] Brown, Marc H. "Perspectives on Algorithm Animation." In Proceedings of *CHI '88 Human Factors in Computing Systems*, pages 33-38, Washington, D.C., May 15-19, 1988.

[Delisle and Schwartz 1986] Delisle, Norman, and Mayer Schwartz. "A Programming Environment for CSP." In Proceedings of *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 34-41, Palo Alto, CA, December 9-11, 1986, Published In *ACM SIGPLAN Notices* 22(1), January 1987.

[Dijkstra 1965] Dijkstra, E.W. "Cooperating Sequential Processes." Technical Report EWD-123, Technological University, Eindhoven, The Netherlands. 1965.

[Eisenstadt and Brayshaw 1987] Eisenstadt, Marc, and Mike Brayshaw. "The Transparent Prolog Machine." Technical Report 21a, Human Cognition Research Laboratory, Open University, Milton Keynes, England. 1987.

[Gaver and Smith 1990] Gaver, William W., and Randall B. Smith. "Auditory Icons in Large-Scale Collaborative Environments." In Proceedings of *Human Computer Interaction — Interact '90*, pages 735-740, Cambridge, U.K., August 27-31, 1990.

[Haibt 1959] Haibt, Lois M. "A Program to Draw Multi-Level Flow Charts." In Proceedings of *The Western Joint Computer Conference*, pages 131-137, San Francisco, CA, March 3-5, 1959.

[Hoare 1978] Hoare, C. A. R. "Communicating Sequential Processes." *Communications of the ACM* 21(8): 666-677, August, 1978.

[Hoare 1974] Hoare, C.A.R. "Monitors: An Operating System Structuring Concept." *Communications of the ACM* 17(10): 549-557, October, 1974.

[Holt and Cordy 1988] Holt, Ric C., and James R. Cordy. "The Turing Programming Language." *Communications of the ACM* 31(12): 1410-1423, December, 1988.

[Isoda et al. 1987] Isoda, Sadahiro et al. "VIPS: A Visual Debugger." *IEEE Software* 4(3): 8-19, May, 1987.

[Knowlton 1966] Knowlton, Kenneth C. *L^6: Bell Telephone Laboratories Low-Level Linked List Language*. Technical Information Laboratories, Bell Laboratories, Inc. 16 mm black and white sound film, 16 minutes. 1966.

[Myers 1990] Myers, Brad A. "Taxonomies of Visual Programming and Program Visualization." *Journal of Visual Languages and Computing* 1(1): 97-123, March, 1990.

[Myers 1986] Myers, Brad A. "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy." In Proceedings of *CHI '86 Human Factors in Computing Systems*, pages 59-66, Boston, MA, April 13-17, 1986.

[Myers et al. 1988] Myers, Brad A. et al. "Automatic Data Visualization for Novice Pascal Programmers." In Proceedings of *The IEEE Workshop on Visual Languages*, pages 192-198, The University of Pittsburgh, Pennsylvania, October 10-12, 1988.

[Reiss 1985] Reiss, Steven P. "Pecan: Program Development Systems that Support Multiple Views." *IEEE Transactions on Software Engineering* 11(3): 276-285, March, 1985.

[Scheifler and Gettys 1986] Scheifler, R.W., and J. Gettys. "The X Window System", *ACM Transactions on Graphics* 5(2): 79-109, April, 1986.

[Sheil 1981] Sheil, B.A. "The Psychological Study of Programming." *ACM Computing Surveys* 13(1): 101-120, 1981.

[Socha et al. 1989] Socha, David et al. "Voyeur: Graphical Views of Parallel Programs." *ACM SIGPLAN Notices* 24(1): 206-215, January, 1989.

[Tufte 1990] Tufte, Edward Rolf. *Envisioning Information*. Cheshire, CT: Graphics Press, 1990.

[Zimmermann et al. 1988] Zimmermann, M. et al. "Understanding Concurrent Programming through Program Animation." In Proceedings of *The Nineteenth ACM SIGCSE Technical Symposium on Computer Science Education*, pages 27-35, Atlanta, GA, 1988, Published In *ACM SIGCSE Bulletin* 20(1), February 1988.

* *

*