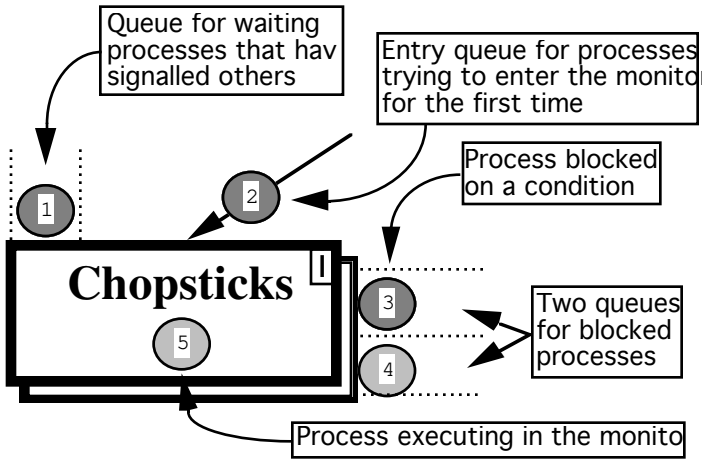

Figure 4: Processes Queuing Around a Monitor

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 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].

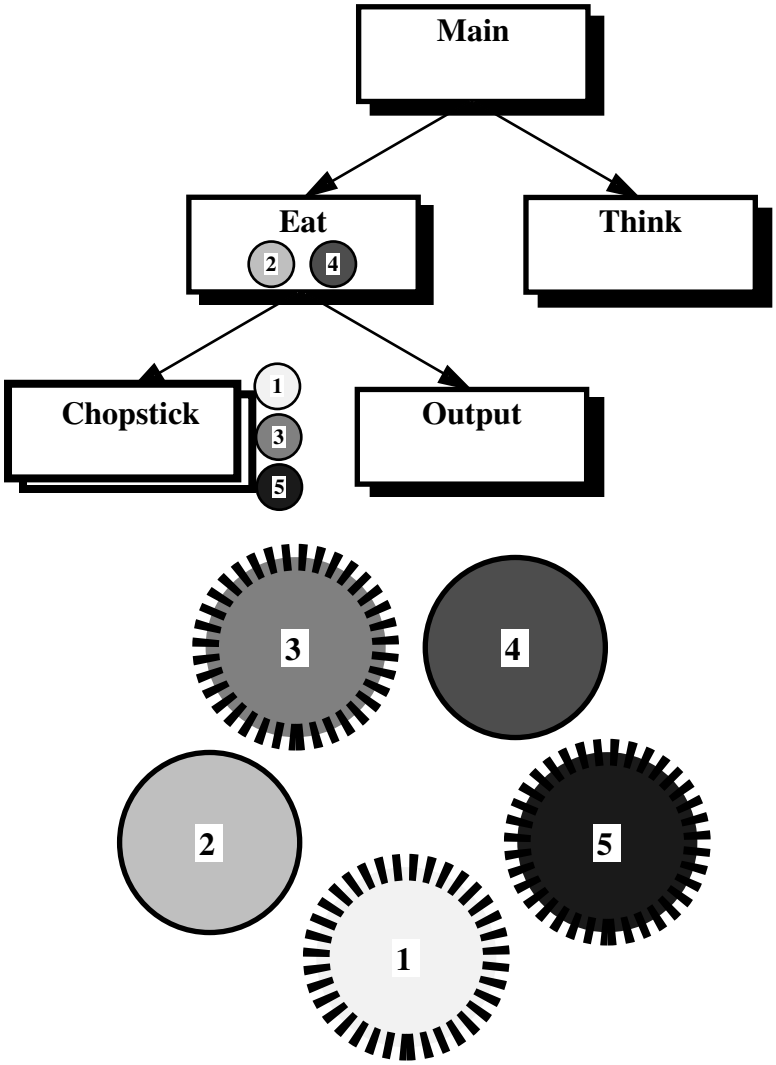

Figure 5: Source-Based and Process-Based Views Positioned By User (Dining Philosophers Problem)

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*.

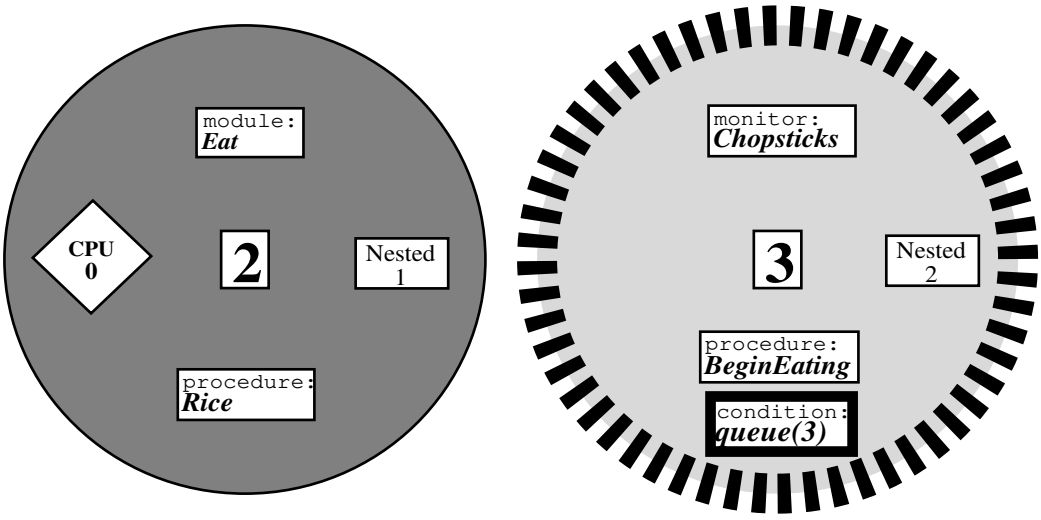

Figure 6: Two Processes in Process-Based View

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 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

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

Table 2: Numerical User Testing Results

"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 *A C M 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.