Visualizing the Execution of Multi-processor Real-Time Programs

Scott Flinn
William Cowan
Department of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3G1

Abstract

This paper describes a monitor designed to provide graphical display of the execution of multi-processor real-time programs. Two design principles, insisting that the monitor should run in real-time and putting no special hooks for the monitor into the operating system kernel, turn out to have far-reaching consequences: the monitor can detect only the state of the program, not its transactions, and any information displayed must be a translation of fast actions within the program into the slower time scale of human perception. The result is a statistical notion of program state. When this type of state is displayed graphically it is discovered to have desirable scaling properties. In addition to the temporal scaling that allows it to mediate between computer and human time scales, it scales well with graphical demands imposed by program complexity.

1. Introduction

Displaying the status of systems of interacting components is probably the earliest widespread application of the graphical display of computer output. Typical examples are the status displays of railroad systems, chemical plants, nuclear reactors and air defence systems. These displays have several characteristics in common. They have minimal interactivity: the display configuration is delivered to users who have little or no ability to change it, and must adapt themselves to it. They use size to deal with structural complexity: display surfaces for large systems may encompass a whole wall. And they display their output in real-time, portraying events that occur on time scales amenable to human perception: trains take minutes moving from one block to another; warning thresholds in process control are set at levels that leave many seconds reaction time before dangerous conditions obtain. Today's multi-processor real-time systems are much more complex, yet programmers usually work without the same level of graphical support. Display constraints are more restrictive than in the older production systems. The display surface is small, rarely larger than the screen of a bit-mapped workstation. Visualization software copes with the limited display space by providing interactive selection of displayed information to the user. Furthermore, events occur on time scales as short as microseconds, but it is usually highly undesirable to slow down the system, which has an important real-time component. We are currently developing a system to be used for visualizing the execution of multiprocessor real-time programs. This paper describes the issues encountered in creating such a system, and some of the solutions we have discovered while experimenting with programs that test components of the complete system.

There are many reasons to determine the behaviour of real-time programs running on multiple interacting processors. For example:

- The programs must be debugged. Conventional debuggers show the behaviour of bugs that are well-localized, but new visualization tools are needed for interactions between high level program components.
- Resource utilization must be monitored, to measure how much is used, when, and by which program components.
- Program execution must be monitored in production environments, to confirm ongoing correct behaviour.
- Visualization can be a form of dynamic documentation. It is particularly useful when demonstrating or teaching system operation to non-programmers.
- Many program development systems create building blocks for program assembly by non-programmers [1]. Execution visualization is an essential support tool in such systems.

Several ways of analysing faulty behaviour in these pro-

grams are not addressed here, since they are better handled by different tools. Two prominent examples are faults in the sequential execution of individual tasks, which are best exposed in task-by-task execution using scaffolding [2] to simulate the interaction with other tasks, and techniques whereby input is recorded so as to play back program execution at varying speeds and granularity. (The latter is the subject of ongoing research by two groups at Waterloo [3, 4].)

Execution visualization and monitoring has been approached from many directions. Most work has gone toward the textual animation of algorithms in which small sections of source code are presented on the screen, and portions are highlighted as they are executed. A modern example is Reiss's PECAN system [5]. There have also been graphical extensions of this basic idea [6, 7, 8]. Subsets of the algorithm visualization problem have also been examined, for example, work of Fumas on viewing large information structures through small virtual windows using fisheye views [9], and of Böcker, Fischer and Nieper for providing tools to be used as a software oscilloscope [10]. Execution visualization has been prominent in parallel, distributed or multi-task debugging. Cheung, Black and Manning have produced a review of this work [11], which includes visualizing of inter-process communication in a message passing environment [12], using instant replay to make parallel programs deterministic[13], and using special purpose hardware to measure performance in real-time [14]. We are unaware, however, of work which attempts to display process communication and activity in real-time without assuming either hardware assistance, or special software assistance from either the kernel or its applications. Any system that does so must develop new techniques for visualizing task execution and inter-task communication. It is specially important for the system to support effective visualization of problems that are unique to program execution in multi-task, multi-processor systems. For example,

- non-deterministic behaviour (especially in the resolution of critical races),
- detection of deadlock and starvation,
- error latency and propagation,
- a very large state space, with much state and event information to be displayed, and
- tightly and loosely coupled program modules (tasks).

Complicating the display problem is the necessity that the system run in real-time, with minimal effect on the behaviour of the program being visualized.

This paper discusses issues that arise in the construction of monitors for visualizing execution of multi-processor real-time programs. Our current knowledge of these issues comes, not from the creation of a complete monitor, but from the creation and observation of monitor components, part of a feasibility study for such a monitor. The next section introduces the basic four stage design of a program monitoring system, including a discussion of constraints influencing the design. Sections 3, 4 and 5 describe the principle monitor stages, emphasizing aspects having to do with the graphical output. Section 6 then discusses some of the important issues of interaction and user control over the monitoring environment. Before going on, however, we include brief definitions of some important terms, necessary because terminology varies in this field. These definitions are based on distinctions made in documentation of the Harmony operating system [15], on which our system is based. A *processor* is a piece of hardware on which programs run. A *task* is a running entity, consisting of code and state. A *program* is the set of all tasks running on a collection of processors. A *meta-task* is a collection of tasks, usually, but not necessarily, closely related. The *system* is the operating system software that provides the program with the abstractions it needs to run. The *state* of a task is the complete set of information that determines its execution context. Often we are interested only in a subset of this information, in which case state is used to describe that information alone. A *meta-state* is a set of states, usually closely related, to be displayed using a single representation. An *event* is a step in a task's execution that changes its state, and usually refers only to the abbreviated state. A *transaction* is an exchange of information between two tasks.

2. Design Constraints

The system discussed in this paper is specifically designed to monitor the execution of multi-task programs running under Harmony. Harmony is a real-time multi-task multiprocessor operating system designed by Morven Gentleman at the National Research Council of Canada for use in embedded microprocessor applications [15]. It has a shared memory and a dynamic task structure, with explicit Create and Destroy primitives. Tasks communicate and synchronize by message passing, using Send-Receive-Reply with blocking Send. We expect to use it for monitoring execution of programs running as part of our experiment workstation. The intent of this workstation is to enable 'non-programmers' to create multi-task programs by assembling experiment components specified by scripts [1]. Effective graphical monitoring of program execution is important for teaching the system as well as for debugging.

Programs to be observed by the monitor run on a variable number of Dy-4 DVME134 boards connected by a VME bus; the monitor itself runs on an independent DVME134 board. Each board consists of an MC68020 processor along with its associated memory and peripherals. A frame buffer is supplied for the monitor's graphical output.

The VME bus is connected by ethernet to a VAX 8600 running Unix, on which programs are compiled and linked prior to downloading [16].

Any monitor interacting with a program at a low level has inevitable dependencies on the environment of the program it monitors: in fact it is common practice to modify system software or even hardware to provide facilities for the monitor to use. Unfortunately such modifications make the monitor very system specific, significantly undermining the generality of results deduced from its performance. To minimize these complications we have accepted two constraints in this work: except for creation of the monitor by FirstUserTask no modifications are made to the kernel or to the monitored program, either code or data structures; and the monitored program is always to run with its execution, and particularly its timing, as unaffected by the monitor as possible. The first constraint forces the monitor to use only a subset of information that is potentially available in the system. The second forces us to deal with timing differences between program and monitor. The constraints restrict what can be monitored. This paper argues that what remains provides a useful and easily implemented capability. The ultimate test, however, lies in extensive experience with the completed monitor, something that lies in the future.

Accepting the constraints has far-reaching consequences, listed below.

1. The monitor must reside entirely on a dedicated processor from which it can read system data structures.
2. The monitor may not write in either program or system memory.
3. The monitor cannot synchronize with events in the program. It determines the state of a program by reading its data structures, and must infer the existence of events or transactions by observing changes in state. In fact, while observation of a state change guarantees that at least one event has occurred, observation of no state change does not guarantee that no event has occurred.
4. The monitor does not necessarily see the program in a consistent state since the acquisition of state information is not atomic with respect to program execution. For this reason display of the measured state may be misleading. It can indicate, for example, the presence of deadlock when none is present. Deciding how to interpret the information gathered and how to display it interacts with monitor timing.
5. Information that changes faster than perceptual integration times of the human observer cannot be displayed. This limitation restricts the monitor to displaying states that persist for seconds. All is not lost, however, since statistical information often persists in time. Suppose a task is in one state most of the time and in a second state the remainder of the time. The proportions can be long-lived even though the persistence of either state is very short. This observation is applicable, since consistency conditions for statistical measures differ from consistency conitions of a single sample. The essential open question is: Are there statistical quantities persisting as long as visual integration times that are useful for program monitoring? Two examples provide prime facie evidence that there are. The first is the utility of profiling tools [17]. The second is the persistence in time of conditions indicating starvation or deadlock. (Programs that accept input are by their very nature indeterminate. In sequential programs without a real-time component, it is often possible to enumerate all logically distinct inputs and test the program against them, making it deterministic for any given input set. For real-time programs such enumeration is impossible, making program execution effectively indeterminate. Interestingly, statistical measurement of program execution is useful because, although there are no program invariants in the usual sense for indeterminate programs, there may well be statistical invariants.)
6. Without special debugging code, the monitor cannot obtain semantic information from the program being monitored. This limitation bounds the level at which program behaviour can be interpreted. In particular, state information from any task is restricted to that information, common to all tasks, which may be extracted from the system asynchronously. If the program introduces task states at a higher level of abstraction, the monitor cannot represent them. Similarly, while message contents might be partially obtained by the monitor, it cannot associate a meaning with the message.

Data acquisition and display is based on a single structure to which all tasks are related. Harmony maintains a simple task structure based on a parent/child relationship in which each child is created by a unique parent. Thus the task structure is a tree rooted at the FirstUserTask. Figure 1 shows a hypothetical task tree spread across four processors, including the monitor processor. Integrating the monitor with the task tree makes it possible to find the root of the tree without using a load map. Task creation and destruction create problems for this scheme. Since they change the tree, data acquisition must be done carefully if spurious information is to be avoided. They also complicate the display requirements, since the monitor must be capable of displaying the result of a task partly existing and partly not existing. (Consistency is further discussed in section 3.)

The monitor is configured into a simple four stage design capable of gathering and displaying state information within the constraints. Collector, condenser and display

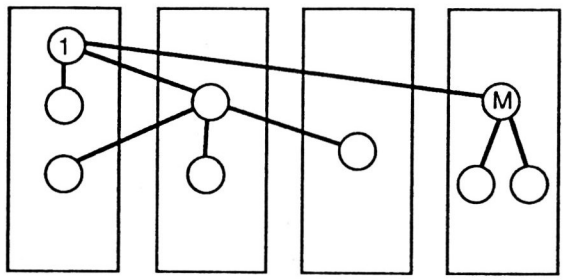

Figure 1. Diagram of a hypothetical task structure. The circles show tasks, the rectangles processors, and the lines parent-child relationships. 1 is the FirstUserTask and M is the root of the monitor.

stages are implemented as functions which, together with necessary local data structures, define a 'module'. Associated with each task is a list of those modules that determine how that task will be displayed. The coordinator is implemented as a separate task which repeatedly builds a local version of the task tree, then applies the module functions for each task in the tree. The collector is the source of the information displayed on the screen. It reads system data structures and records information locally to be processed by the corresponding condenser. The condenser is then invoked to provide the summarizing of data needed to mediate between the microsecond time scale of data collection and the several second time scale of the display. The final display stage converts the collection of summarized information produced by the condenser into graphical images for display to the user.

Note that all four of the stages can be modified on the basis of interactive input from the user. The components of a module are discussed in detail in Sections 3 to 5. User interaction is then discussed in Section 6.

3. Collection of State Information

The collection of state information is relatively straightforward. Each collector is responsible for some of the state information for each task in the tree. Typical parts of the task state are

- execution state (running, ready, blocked),
- position in various queues (ready queue, message queue, etc.),
- current correspondent,
- waiting correspondents,
- priority,
- i/o connections, and
- owned or locked resources.

For each task in the tree the coordinator calls each of the associated collector functions, which determine the task state from the system data structures. This information is then processed by the condensers. The set of operations is cyclic, with a frequency determined by the user. The sampling frequency controls the amount of interference between the monitor and the program: at its fastest we are able to use about half the memory bandwidth for sampling, greatly affecting the real-time response of the monitored program but missing very few events; contrarily there is no limit on the low side, except that the program is seen at a coarser granularity.

There are two features to be noted. First, to maximize the portability of the monitor it is important to isolate peculiarities of the kernel data structures. They are represented as subordinate functions of the collectors. The coordinator/collector model makes adding or deleting a new data field as simple as adding a module whose collector knows the location of the data relative to the task descriptor. Similarly, changes in the task structure, such as a separate task tree for each processor, requires changes limited to the coordinator. The requirement that the monitor should be able to read the data structures of the system, however, cannot be changed. In systems without a shared memory space, as Harmony has, kernel modifications are needed to allow the collectors to operate.

A more complex issue is creation and destruction of tasks. Because the monitor does not synchronize with the program, it is possible for a task to be destroyed between its detection by the coordinator and subsequent accesses by collectors. If the collector checks the validity of the task at the end of its access, it can throw away information that may be corrupted. When a task is in the process of being destroyed it is possible for some collectors to access it validly while others fail to do so. Since each collector report is a small part of the statistical display generated by the monitor, however, any perturbation is minor. Tasks that are missed by the collectors on a given cycle because they are created between the coordinator access and the collector access are also a minor perturbation. It is even possible, though unlikely, for a task to be missed altogether when the monitor is run with a long cycle time (very low interference) and the create/destroy cycle is very short. At all times the user must be aware that the monitor is sensitive to a granularity of computation comparable to the cycle time and that events of smaller granularity may be missed.

4. Condensation of State Information

The condenser accepts information from the collectors and processes it into a form suitable for display. As much as possible of the temporal processing is concentrated there, while other aspects of the display are handled by the last

stage. Thus, in general, the condenser accepts input at a rate it cannot control, sorts and summarizes it, then puts out the summaries.

Two aspects of the condenser function influence the interactive quality of the monitor. The first is the creation of data summaries. Human perceptual systems cannot follow displays that change their contents much faster than several times a second, during which time there can be thousands of samples taken by the collectors. Since different types of data are suited to different types of summary, a wide range of summarization techniques is needed. Here are a few examples.

- Quantitative summaries, like mean, standard deviation, minimum and maximum, for data that are numerical, such as memory in use or number of transactions detected.
- Proportions of samples in a given category for data that is qualitative, such as execution state. Such proportions estimate the amount of time a task spends in a given state.
- For properties like memory utilization the user must get an impression of change in time. Changes can happen so fast that the user cannot see contrasts between past and present. In such a case, not the value of the current sample, but a weighted average of the last sample and the current one should be reported. The weighting determines the rate at which the past dies away.

The second is the detection of events and transactions. Since the monitor is not synchronized with the program, events and transactions are inferred from observed changes in state. This inference is made by the condenser. Since other systems have different states and different events it is important for portability to embody the rules in a data-base. Since it is impossible to be sure that all events and/or transactions of a given type have been detected, the values determined by the condenser are a lower limit. Note that some events or transactions, like interrupts or message exchanges, which are usually completed in less than a millisecond, are effectively instantaneous on the time scale of the user. They should be reported as numbers of events per display period, reported either raw or as a moving average. Other transactions, like the transfer of very large blocks of data, can be extended on the time scale of the user. In Harmony many transactions have a set of intermediate states occurring between initial and final ones. The condenser can use them to provide the display controller with the information needed to support displays that indicate the progress of the program through a transaction.

5. Graphical Techniques

The graphical interface of a program like the execution monitor is very complicated. In the interest of preserving space, this discussion is limited to aspects of the monitor interface that we consider to be unique. The first is attentional selection. The display is intended to be as rich as possible in information, with the user attending selectively to the subset of it that is of interest at any given moment. A road map is a useful analogy. It contains far more information than an observer can take in at a glance, or even as the result of concentrated study, but the information is organized so that it is easy to attend only to a small subset: the roads between Toronto and Ottawa, for example, or rivers running into Lake Ontario. Because the monitor display is animated, effective organizational principles are more complicated than for a road map, which is static. Another obstacle is the limited resolution of current display surfaces compared to printed material. Finally, because most aspects of the display must be based on monitor-supplied defaults, the monitor itself must recognize structures that should be visually associated and choose presentation methods that provide the visual association effectively.

A second novelty of this program is that the information displayed is almost exclusively statistical. This information is nominal or numeric. Task state is an example of nominal information. The condenser generates numbers that are, in essence, the proportion of time spent by a task in each state. Consider the problem of presenting this information, taking colour as the modality for the sake of concreteness. One option is to produce a single percept that is an appropriate average. If the task has only two states, READY and ACTIVE for example, code one as yellow, the other as green. Then states consisting of mixtures of the two can be represented as intermediate yellow-green colours with the proportions of yellow and green determined by the time spent in each state during the sample period. Suppose, however, that another state is added, BLOCKED for example, which is coded as red. There is now no unambiguous interpretation of a yellow colour. It might be 100% READY and it might be 50% ACTIVE and 50% BLOCKED. Different colours might be chosen to make the three state display unambiguous but, since colour vision is three dimensional, ambiguity is unavoidable for four or more states. A better solution divides the task symbol into regions, coloured by state and proportional in size to the time spent in the state, as in figure 2.

Note that this solution provides some useful scale invariance and in doing so subsumes the rejected mixing idea, since viewing the display from a distance produces a percept that is the mixture of the colours. Another feature of this solution is that it effectively limits the number of displayed states to something like ten [19]. In Harmony, the number of task states greatly exceeds ten, but many states are closely

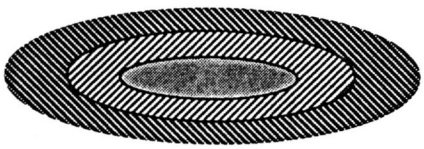

Figure 2. Possible colouring of a task state, using textures to stand for colours. This particular task symbol illustrates a possible problem. Presumably proportion is associated with area in the symbol, but humans do not make size judgments that are linear with area. Thus, precise estimation wold require a differently structured symbol.

related. They are grouped into meta-states for display by a single colour. The meta-state solution is important regardless of stimulus dimensionality, since comprehension of a large number of states is a significant problem. It is also important in task grouping, as explained below.

In the example above hue was chosen to display nominal information, a suitable choice [20, 21]. When numeric information, such as the number of transactions between two tasks, is to be displayed, size is a better variable. Suppose, for example, that transactions between two tasks were to be displayed as an arrow linking the tasks. Making the width of the arrow proportional to the number of transactions detected is an effective display technique. An interesting feature of this display solution is that it requires the tasks to be in close spatial proximity. Thus, tasks must be grouped together for display. Task grouping can be done according to a variety of criteria, position in the creation tree, processor on which the task runs, transaction rates or even according to the logic of the program, which is implicitly defined by the user. Task display is greatly simplified when these criteria lead to the same grouping, and it is very likely true that ease of task display is closely related to ease of programmer comprehension.

As the monitored program becomes more complicated the display can be simplified by collapsing well-structured groups of tasks into a meta-task. A meta-task has simple transaction properties, but its state space is the Cartesian product of the states of its component tasks. Thus, explicit state display is not possible. Fortunately there are many types of higher order statistical summary that can be created from the states of the individual tasks. Well-chosen summaries, which might be created by dithering the meta-task symbol, can have the useful property of resembling scaled verions of the task group, thereby providing useful visual continuity under scaling.

The ability to recognize objects easily at different scales is a very useful property of the visual system, one that must be utilized when the information to be displayed begins to exceed the capacity of the display medium. Sampling is an information gathering technique that scales well in time, since most statistical measures change little as sampling rates change. Similarly, summary statistics like the mean amount of each state in a meta-task statistical state can be made to scale with program complexity. Experimentation with our monitor will demonstrate whether or not there are measures of program performance of this type that are useful to the programmer.

The effects of differing time scales is also important when animating processes that are extended in time. In one of our tests, transactions were animated to test the effectiveness of animation at giving the impression of information moving from one task to another. In fact, the result was just the opposite: the animation, which consisted of small comets moving from one task to another, confused users. Why? For motion to be visible the animation had to be extended over about a second, during which time the comet was displayed several times at intermediate positions. Harmony message passing, on the other hand, consists of three transactions, a send, a receive and a reply, often extended over less than a millisecond. Very quickly the temporal sequence of the transactions was lost as the monitor fell behind, and the comets created uninformative confusion. The result is obvious once time scales are considered. Harmony message passing is normally instantaneous to the human visual system. On the rare occasions when it takes times the order of a few seconds it is important to give the user the impression of progress through a sequence of states extended in time. Cooperation between the condenser and the display is needed to identify sequences that should be animated so as to show them appropriately to the user.

6. Interaction

Providing interactive control of monitor operation is, at present, quite unexplored. The user's job controlling the monitor is difficult for two reasons.

- The monitor operates under a variety of constraints, on communication, processing and display. When manipulation of a variable hits a constraint the situation is almost always confusing for the user, especially so when he is operating in a control space of high dimension.
- The number of potential choices for a user is very high.
- The monitor should present the user with a display that is right enough that only tuning is needed.

It is important to allow the user to interact with all three components of the monitor. Interaction with the first two stages is sparse and easy to implement: the intensity of data collection by the collectors must be adjustable, and the summarization modes used in the condenser must be controlled. Management of the display is a more difficult issue. In ac-

cord with our intention of presenting a maximum of information, allowing the user's visual system to select among it, we have so far concentrated our attention on the creation of useful defaults. All the same two important issues of interaction are worth mentioning.

- Effective meta-task control is essential. Interactive control must cohere effectively with other grouping dimensions, such as creation trees and transaction intensity, as mentioned above. It must be possible to open and close meta-tasks, and to turn on and off graphical indications of current grouping strategies used by the monitor.
- Interaction functionality of a 'wait for state' variety is an important visualization tool, existing as breakpoint setting in hardware such as logic analysers, and in software in all types of debuggers. In an environment where state is non-deterministic because it is statistical, 'wait for state' functionality must be generalized. One generalization is obvious, since some interesting events, like deadlock, produce states of prolonged duration. Other definitions of state, like interrupt service routines running more frequently than input data rates, are also likely to be useful.

In sum, interacting with display modalities remains largely unexplored. Such fundamental issues as direct vs indirect manipulation are not obvious in an environment as complex as this one.

7. Conclusions

We have described a monitor we are building for the interactive visualization of the execution of muti-task real-time programs. Two simple principles, running the monitor at the speed of the monitored program and non-modification of the program or the kernel of the operating system under which it runs, have surprisingly far-reaching consequences, including the alteration of task state to a statistical quantity. Statistical state coincides with problems matching the user's time scale to the time scale of the executing program. Since the user can only perceive states on a much longer time scale than program operation, visualization techniques concentrate on configurations that are prolonged in time. Our experience in creating graphical techniques suitable for displaying statistical state is still in its infancy.

Even at this early stage of our project it is possible to derive some concrete conclusions. One concerns task structuring. The multiplicity of criteria for the visual creation of task groupings on the monitor display reveals a multiplicity of criteria on which task structuring can be performed. It is important for programmers to restrict themselves to task structuring that is consistent across criteria if they want to produce programs with a structure that is easy to grasp. For example, analysis of remote delay [22] demonstrates that dynamic task structure is an essential part of multi-task systems. Task grouping considerations revealed by the monitor show that some program locations are to be preferred over others as sites for task creation and destruction, at least if structural simplicity is a programming objective.

Another conclusion concerns the nature of statistical state. As programs become more complex, the ability to relate activities that occur at different program granularities becomes more and more important. Visual tools are clearly important for this objective, particularly since the visual system has highly evolved capabilities for making correlations between views of objects at different scales. Interestingly, problems creating good visual representations of statistical state seem to be closely related to scaling problems. In the monitor they arise in the temporal domain, but graphical display considerations show them to be important spatially as well. Further study of them is likely to provide important tools for graphical display of complex systems.

Finally it is worth mentioning that the techniques discussed in this paper are likely to have application outside the domain of real-time systems. Consider, for example, probabilistic algorithms [23]. Because they are non-deterministic they cannot, by definition, have program invariants. By techniques similar to the ones discussed in this paper it is possible to define statistical invariants for them, providing possibilities for verification and insight into quanitities that are interesting when displaying their execution. Such algorithms use random numbers that are pseudo-random in most cases. Thus, at the granularity of the internal operation of the random number generator such algorithms become deterministic; at a larger granularity they are non-deterministic. Most programs, particularly those involving time and/or external input in an essential way, are similar in being deterministic at small granularities and non-deterministic at large granularities. Thus, the issues discussed in this paper are likely to have ever wider applicability as the increasing size and complexity of programs forces us to understand them at levels that are more and more removed from the actual machine on which they execute.

8. Acknowledgements

This research was supported in part by the National Science and Engineering Research Council of Canada and by Digital Equipment Canada. The authors also wish to thank Kelly Booth and Marceli Wein for many helpful discussions.

9. References

[1] L.R. Bartram, K.S, Booth, W.B. Cowan, J.D. Morrison, and P.P. Tanner, "A System for Conducting Experiments Concerning Human Factors in Interactive Graphics", Proc. Graphics Interface 1988, pp. 34-42.

[2] J. Bentley, More Programming Pearls, Addison-Wesley: Reading, 1988, pp. 27–36.

[3] W.-H. Cheung, Process and Event Abstraction for Debugging Distributed Programs, University of Waterloo, Department of Computer Science, Ph.D. Thesis, 1989.

[4] M. Gauthier, Visualizing the Output of the Psychology Workstation, University of Waterloo, Department of Computer Science, Masters Essay, 1990 (in progress).

[5] Steven P. Reiss, "PECAN: Program Development Environments that Support Multiple Views", IEEE Trans. on Software Engineering, SE Vol. 11, No. 3, March 1985, pp. 276-285.

[6] Ronald M. Baecker, "An Application Overview of Program Visualization", Computer Graphics, Vol. 20, No. 4, July 1986, p. 325.

[7] Marc H. Brown, "Perspectives on Algorithm Animation", Proc. ACM SIGCHI'88 Conf. on Human Factors in Computing Systems, pp. 33-38.

[8] Marc H. Brown, Robert Sedgewick, "A System for Algorithm Animation", Computer Graphics, Vol 18, No. 3, July 1984, pp. 177-186.

[9] George W. Fumas, "Generalized Fisheye Views", Proc. ACM SIGCHI'86 Conf. on Human Factors in Computing Systems, pp. 16-23.

[10] Heinz-Dieter Böcker, Gerhard Fischer and Helga Nieper, "The Enhancement of Understanding through Visual Representation", Proc. ACM SIGCHI'86 Conf. on Human Factors in Computing Systems, pp. 44-50.

[11] W.H. Cheung, J.P. Black, E. Manning, "A Study of Distributed Debugging", Research Report CS-88-44, Faculty of Mathematics, University of Waterloo, Waterloo, Ont., October 1988.

[12] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger, "Monitoring Distributed Systems", ACM Transactions on Computer Systems, Vol. 5, No. 2, May 1987, pp. 121-150.

[13] T.J. LeBlanc, J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", IEEE Trans. on Computers, Vol. C-36, No. 4, April 1987, pp. 471-482.

[14] Dieter Wybranietz and Dieter Haban, "Monitoring and Performance Measuring Distributed Systems During Operation", ACM 0-89791-254-3/88/0005/0197, pp. 197-206.

[15] W.M. Gentleman, "Using the Harmony Operating System", Tech. report NRCC-ERB-966 Division of Electrical Engineering, National Research Council of Canada, December, 1983.

[16] K.S. Booth, W.B. Cowan, D.R. Forsey, "Multitasking Support in a Graphics Workstation", Proc. 1st International Conference on Computer Workstations, (November, 1985) pp. 82-89.

[17] J. Bentley, More Programming Pearls, Addison-Wesley: Reading, 1988, pp. 3–13.

[18] J. M. Chambers, W. S. Cleveland, B. Kleiner and P. A. Tukey, Graphical Methods for Data Analysis, Duxbury Press: Boston, 1983.

[19] R. M. Boynton and C. X. Olson, "Locating the Basic Colors in the OSA Space", Color Research and Application, 12, 1987, pp. 94–105.

[20] W. B. Cowan, "Color psychophysics and display technology: avoiding the wrong answers and finding the right questions", Image Processing, Analysis, Measurement and Quality, Gary W. Hughes, Patrick E. Mantey, Bernice Rogowitz, Editors, Proc. SPIE 901, 1988, pp. 186–193.

[21] C. Ware and J. C. Beatty, "Using Color Dimensions to Display Data Dimensions", Human Factors, 30(2), 1988, pp. 127–142.

[22] B. Liskov, M. Herlihy and L. Gilbert, "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing", ACM Symposium on Principles of Programming Languages, 1986, pp. 150–159.

[23] D. J. A. Welsh, "Randomized algorithms", Discrete Applied Mathematics, 5, 1983, pp. 133–145.