The Automatic Animation of Concurrent Programs

Blaine A. Price[1]
Ronald M. Baecker

Dynamic Graphics Project
Computer Systems Research Institute
University of Toronto, CANADA
M5S 1A1
Telephone: +1 (416) 978-6619
Fascimile: +1 (416) 978-4765
e-mail: rmb@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.

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

[1]Author's current address: Human Cognition Research Lab, Open University, Milton Keynes MK7 6AA, UK e-mail: ba_price@vax.acs.open.ac.uk

Reprinted from *Proceedings of the First Moscow International HCI Workshop*, 5-9 August 1991, Moscow, USSR. Publisher: The International Centre for Scientific and Technical Information, pp. 128-137

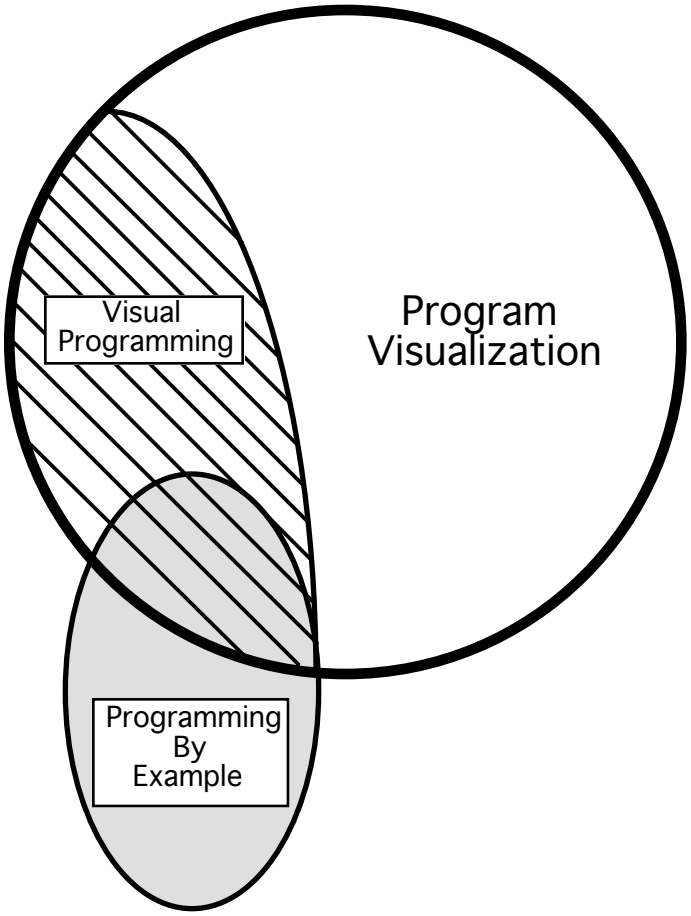

Figure 1: The Relationship Between Program Visualization, Visual Programming, and Programming By Example

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. 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[1] 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.

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

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 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

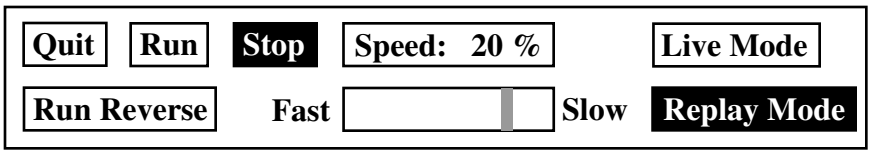

Figure 2: The *Paradocs* Control Panel (program stopped in replay mode)

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.

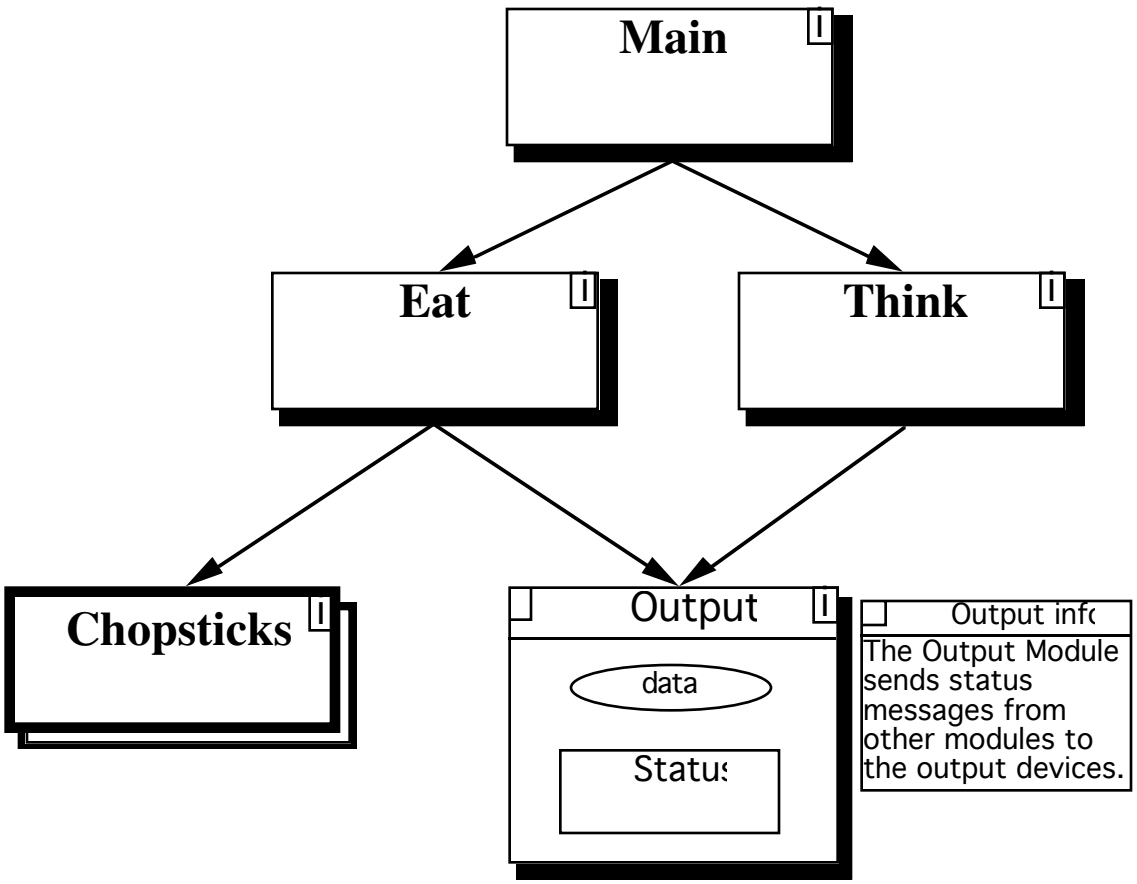

Figure 3: The *Paradocs* Static Source-Based View