This is a hypertext multimedia version of the paper:
Price, B.A., Baecker, R.M., and Small, I.S. ‘‘A Principled Taxonomy of Sofware Visualization’’
Journal of Visual Languages and Computing **4**(3):211-266.

A Principled Taxonomy of Software Visualization

by Blaine A. Price **+***, Ronald M. Baecker *****, and Ian S. Small ^*****

+ Human Cognition Research Lab, The Open University, Milton Keynes MK7 6AA, Britain

***** Dynamic Graphics Project, CSRI, University of Toronto, Toronto, ON M5S 1A1, Canada

^ Advanced Technology Group, Apple Computer, Inc., Cupertino, CA 95014, USA

Abstract

---

1. Introduction

2. Definitions

3. Other Taxonomies and Surveys

3.1. A Brief History of Software Visualization

3.2. Current Uses of Software Visualization

4. Twelve Systems

4.1. Sorting Out Sorting

4.2. BALSA

4.3. Zeus

4.4. TANGO

4.5. ANIM

---

1. Introduction

A well founded taxonomy can further serious investigation in any field of study. A common language or terminology facilitates communication about ideas or discoveries. Taxonomies provide this common language and allow new discoveries to be identified and catalogued. They also show where an apparently new discovery is a refinement or variation of something else. In the natural sciences, taxonomies (such as the periodic table of elements) have also served to predict where new discoveries will be made. An important feature of a taxonomy is that it allow for expansion; if a new branch of a field is discovered or invented it must fit smoothly into the taxonomy without requiring a re-ordering of all of the other items. If a particular area later warrants closer study then a finer subdivision must be allowed. Thus, a taxonomy must have a principled derivation, for an ad hoc approach invites chaos and frequent problems in categorizing new items.

In this paper, we present a new taxonomy for systems involved in the visualization of computer software. We seek to provide a detailed "road map" of the work accomplished so far by identifying six broad categories of characteristics derived from an accepted model of software and by filling in the observed characteristics in each category. We describe twelve systems in detail and then apply the taxonomy to them in order to illustrate its application and to show how the systems span the taxonomy.

Many of the characteristics require a subjective evaluation to rank systems relative to one another. Each ranking is based on our understanding of the systems and in some cases on our personal opinion of their relative performance; this is the weakest point in our taxonomy and more rigorous methods are required for evaluation. In the discussion we look at the performance of each system in each category and comment on its contribution to the field. We also discuss how the fields of cognitive science and software psychology have been underutilized by researchers building systems and suggest a number of ways in which this work can make a contribution. We conclude with a research agenda for the 90's.

2. Definitions

Phrases like *visual programming, program visualization*, and *algorithm animation* have been defined and used many times in the literature, yet future clarification is still necessary. The use of the word "visual" in "visual programming" can be misleading if one only considers the common definitions of the word "visual". In the Oxford English Dictionary (Simpson & Weiner, 1989) (p. 699), the first six of the seven definitions of "visual" relate to information gained from the use of the human eye, while the seventh suggests the conveyance of a mental image. It is this latter definition that applies to "visual languages" or "visual programming" since a mental image can be formed as a result of input from any of the senses. All programming (at least in the last thirty years) is "visual" in the common sense of the word, since it involves programmers reading printed code (with their eyes). Virtually all modern programming is also at least weakly visual in the "mental image" sense, since human programmers do not read code serially as a stream of bits in the way an interpreter or compiler does. Even programmers who use a simple VT100-style terminal in a single colour, fixed-pitch font can get a mental image that aids comprehension from the appearance of the indenting in their code (Norcio, 1982; Miara, *et al*., 1983) and the relative sizes of code blocks (cf. Baecker and Marcus's (1990) program maps).

Because it contains the root word "visual", "visualization" is often considered to be restricted to visible images (hence the coining of words like *auralization*). In fact, its primary meaning is "the power or process of forming a mental picture or vision of something not actually present to the sight" (Simpson & Weiner, 1989) (p. 700), so a visualization can result from input from any combination of the human senses. *Program visualization* has been defined by several authors, but the general consensus is that it is the use of various techniques to enhance the human understanding of computer programs, while *visual programming* is the use of "visual" techniques to *specify* a program in the first place. The problem with the term "program visualization" is that it becomes ambiguous when considered in the context of its constituent parts. *Algorithm visualization* (or *animation*) is understood to be the visualization of a high-level description of a piece of software which is in contrast to *code* or *data visualization* (which are collectively a kind of program visualization) where actual implemented code is visualized. We prefer the term *software visualization* (hence SV) to include all of these because it eliminates the ambiguity and covers all of the software design process from planning to implementation. It also includes software composed of multiple programs.

We define SV as the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software. Considered strictly, this means that the representation of a program written in a visual language is a kind of SV, although it would be considered a weak version since it is designed to facilitate *specification* rather than *understanding*. In this paper we will only consider systems employing intentional SV and thus no visual programming systems are mentioned. Figure 1 is a

Venn diagram showing how each of the terms in the literature fit together under our definitions. We note that in some cases the boundaries in this diagram become blurred, such as when a very high level program specification is visualized or when the division between instructions and data becomes indistinguishable.

Some other terms require definition within the scope of this paper. We use the term *programmer* to refer to the person who wrote the original program that is being (or going to be) visualized. Programmers may not have known that their programs were going to be visualized when they wrote them. Another kind of programmer is the *SV software developer* who wrote the system used to create the visualization. We use the word *visualizer* to mean the person who created the visualization from the original program using the SV system and the word *user* to mean the person using the visualization to understand the original program. A single person may have more than one of these roles.

3. Other Taxonomies and Surveys

The best known taxonomy and survey is probably that of Myers (1986) which has been updated twice (Myers, 1988; Myers, 1990). In his first paper, Myers provided an excellent differentiation between Programming by Example, Visual Programming, and Program Visualization. In his latest paper, he looked at nineteen SV (PV in his nomenclature) systems and classified them along two axes: their level of abstraction (from showing code or data to showing algorithms) and the level of animation in their displays (whether static or dynamic). This resulted in a 2x3 grid with several systems falling into multiple categories. This taxonomy is a good starting point but we believe that the variety of systems, goals, and technology available demands a more thorough approach. Myers's axes are certainly some of the most important characteristics of an SV system, but his section on "General Problems and Areas for Future Research" reveals a number of characteristics which are worthy for distinguishing existing systems.

Stasko and Patterson (1992) introduced scaled dimensions in their four-category taxonomy covering aspect, abstractness, animation, and automation. Although the title of Shu's (1988) book is *Visual Programming*, she discusses SV systems in the early chapters and her chapter and sub-heading breakdown indicates the same basic characteristics as Myers (data vs. code visualization and static vs. animated). Chang's (1990) book surveys a large number of VP systems, many of which present visualizations of software. Other taxonomies and surveys with related work include Glinert's (1990b; 1990a) extensive survey of VP systems (which includes some SV) and Cypher's (1993) comprehensive study of Programming by Demonstration (formerly Programming by Example) systems, some of which use visualization.

In choosing to expand these taxonomies we saw two important problems: to choose characteristics in a systematic way, and to allow for future expansion and revision as the field changes. We address the latter problem by building the taxonomy hierarchically so that new nodes could be added easily. We address the former problem by deriving the top level of the hierarchy from a basic model of software so that as characteristics are discovered they can be placed in the appropriate group to determine if they are in fact new.

3.1. A Brief History of Software Visualization

The importance of visual representations in understanding computer programs is by no means new.

Goldstein and von Neumann (1947) demonstrated the usefulness of flowcharts, while Haibt (1959) developed a system that could draw them automatically from Fortran or assembly language programs. Knuth (1963) developed a similar system which integrated documentation with the source code and could automatically generate flowcharts. Although early experiments cast doubt on the value of flowcharts as an aid to comprehension, recent results are more encouraging (Scanlan, 1989).

A different approach was taken with Knowlton’s (1966a; 1966b) films, which showed list manipulation in Bell Lab’s L[6] language (Knowlton, 1966c). This work was the first to use dynamic as opposed to static techniques and the first to address the visualization of data structures. Baecker’s (1968) debugger for the TX-2 computer produced static images of the display file, but it was live and interactive. Baecker continued this work in a pedagogical direction which resulted in systems for showing data structure abstractions in running programs (Baecker, 1975) and eventually in the film *Sorting Out Sorting* (Baecker, 1981).

The 1970’s also saw a return to flowcharting with the development of Nassi-Shneiderman diagrams (Nassi & Shneiderman, 1973) to counter the unstructured nature of standard flowcharts. Roy and St. Denis (1976) then developed a system for automatically generating Nassi-Shneiderman diagrams from source code through a specialized compilation process.

The remaining development from the 1970’s was pretty-printing, a term coined by Ledgard (1975) to describe the use of spacing, indentation, and layout to make source code easier to read in a structured language. Many systems for automatic pretty-printing were developed, such as NEATER2 (Conrow & Smith, 1970) for PL/I and Hueras & Ledgard’s (1977) system for Pascal. The techniques have proved relatively simple; however recent extensions to this work have used computerized typesetting and laser printing to provide a much improved presentation of source code. This has ranged from a simple utility to change the style of the font depending on keywords (BSD Unix Distribution, 1988) to the SEE Program Visualizer (Baecker & Marcus, 1990) which automatically takes a set of C programs and formats a "program book" out of them. Knuth’s (1984) WEB system is similar, but combines the documentation and program in one document using a markup language.

The 1980’s saw the beginning of modern SV research with the introduction of the bit-mapped display and window interface technology. The most important and well known system of this era was BALSA (Brown & Sedgewick, 1984b), followed by Balsa-II (Brown, 1988b), which allowed students to interact with high level dynamic visualizations of Pascal programs. Many prototypes and production systems using modern human-computer interface technology have been developed since (see the next section for more detail on a number of systems or the work of Myers, Shu, Chang, or Glinert cited above for a larger sample).

3.2. Current Uses of Software Visualization

The motivation for building "visual" systems (both SV and VP) has been well argued in the literature (see, for example, the discussion of the "dual brain" in the introduction of Shu’s (1988) book or the introductions in any number of the other systems cited) so we will not repeat it here. While newer programming environments have begun to make use of more visual techniques to display information, the majority of professional programmers we have observed still rely on the old glass teletype technology: they may use multiple-scrolling windows on large-screen workstations, but by and large they still edit their programs in single colour fixed-pitch text and debug their programs using conventional debuggers. Even the latest workstation programming environments simply provide "window dressing" on conventional debugging tools like Unix's `dbx`. Modern CASE tools, such as "Software Through Pictures" from Interactive Development Environments, Inc., employ static diagrams (see (Martin & McClure, 1985) for detailed examples). These aid in structured analysis and design, but do not use any kind of run-time model or animation.

Why is SV technology not being used and why are the new visual systems not being widely adopted? One likely answer is that software engineers (and their employers) have not seen demonstrable gains in using this technology. It is clear that if SV systems are to make a contribution to software engineering then solid results proving their benefits will be necessary. In the following sections we map the areas that have been explored so far and suggest where continued research will benefit professional software engineers.

4. Twelve Systems

It is easier to understand any taxonomy if one has a number of familiar examples to test against it. In the following sections we give a brief explanation of eleven different SV systems developed in university/corporate research environments and one commercial workstation programming environment with SV features; we will refer to each of these systems later as we explain our taxonomy. We chose the research systems either because of their historic importance or because they illustrate a diversity of approaches (or both). These systems do not include all the historically important SV systems nor do they completely span the space of our taxonomy, but they do serve as concrete reference points for mapping the taxonomy into familiar examples.

4.1. Sorting Out Sorting

The first major SV work of the 1980's was the motion picture *Sorting Out Sorting* (SOS) (Baecker, 1981), produced at the University of Toronto. This 30 minute, colour, narrated educational film uses animated computer graphics to explain how nine different sorting algorithms manipulate their data.

SOS begins by introducing the concept of sorting data and goes on to explain three insertion sort algorithms (linear insertion, binary insertion, and shell sort) followed by three exchange sort algorithms (bubble sort, shaker sort, and quicksort) and finishes with three selection sort algorithms (straight selection, tree selection, and heap sort). At the conclusion, it shows a race of all nine algorithms running in parallel on large data sets. The film introduces each algorithm with an example showing how it affects on the order of ten to twenty data items. The data items are typically represented by blue or green rectangles with each having a different height. When one or more data items are being considered by the algorithm, they are highlighted and when an element has reached its final (sorted) position it turns red. At the end of each section, the film compares each set of three algorithms running in parallel on large data sets.

After explaining all nine algorithms, SOS addresses the efficiency of each by comparing the O(*n[2]*) sorts with the O(*n*log(*n*)) sorts. It illustrates the speed differences by showing each algorithm working on a data set of twenty-five hundred random elements. In this representation, the data are represented by tiny yellow dots on a graph, where the horizontal coordinate represents the element's current position in the data set and the vertical coordinate indicates the value of each element. When an element reaches its final (sorted) position along the main diagonal it turns red. Thus, unsorted data appear as a yellow cloud and sorted data appear as a diagonal red line. In the race of the nine algorithms (each running in parallel on processors of equal speed), quicksort and tree selection sort complete in less than a minute while the bubble sort takes nearly an hour (which is obviously not shown to completion). Figure 2 shows a portion of the race.

Today SOS is sold commercially as a videotape and it is used widely for introductory computer science teaching at the secondary and post-secondary levels, although no formal empirical evaluations have been performed.

4.2. BALSA

The first major interactive SV system was the Brown University Algorithm Simulator and Animator (BALSA) (Brown & Sedgewick, 1984b; Brown & Sedgewick, 1985; Brown, 1988a). This pioneering work has become the benchmark against which all subsequent SV systems have been measured. BALSA was more than a research prototype: it evolved from a principled design and was in production use for years both as teaching tool used by hundreds of undergraduates and as an aid to algorithm design and analysis (Brown & Sedgewick, 1985).

BALSA was one of the earliest systems to take advantage of large screen graphics and windowing based personal workstations. Installed in the fall of 1983 on a set of fifty Apollo workstations in a purpose-built lecture theatre/laboratory, it allowed the instructor to give a running commentary on the prepared graphical animation running on each student's machine. Students could control these scripted animations (stop, start, speed control, replay, as well as standard window control, panning, and zooming). Several of the animations could also be played backwards. The entire set of scripts was integrated with the textbook *Algorithms* (Sedgewick, 1983) and BALSA was used by students in three undergraduate courses (Brown, Myrowitz, & van Dam, 1983; Brown & Sedgewick, 1984a).

BALSA itself was written in C but the algorithms that it animated were usually coded in Pascal. It supported multiple simultaneous views of each running algorithm and it was the first system that could show algorithms racing with each other on the same display. The contents of each view window depended on what the visualizer had provided and could not be changed by the user. A code view was usually provided: it showed a pretty-printed listing of the current procedure with the current line highlighted. If another procedure was called then its window was stacked on top of the calling procedure. Views of data structures, however, were often far more enlightening and could range from the "dots" or "sticks" views of *Sorting Out Sorting* to complex graphs, trees, or computational geometry abstractions. BALSA was able to display multiple algorithms running simultaneously as well as multiple views of the same data structure.

The BALSA visualizer built animations by starting with a standard Pascal implementation of an algorithm and annotating it with calls to BALSA's "Interesting Event" manager at points where the code changed interesting data structures or entered/exited a subroutine. The visualizer could then build a view to respond to the event information by modifying an existing view from the BALSA library or by building a completely new view using the built-in graphic primitives. The final step would be to specify the input generator which provided valid input for the program. Users could run and manipulate the completed animation and record their actions in scripts which others could replay.

The initial versions of BALSA ran in black-and-white on Apollo workstations. Brown's subsequent version, Balsa-II (Brown, 1988b; Brown, 1988a), ran in colour on Apple Macintosh personal computers and allowed rudimentary sounds to be associated with events. Synchronized multiple-algorithm displays which provide "algorithm races" like those in *Sorting Out Sorting* are also within the scope of Balsa-II. Although no formal empirical evaluations were conducted, the authors reported on their experiences using BALSA in undergraduate teaching (Brown, Myrowitz, & van Dam, 1983; Brown & Sedgewick, 1984a). Figure 3(a) shows a still from an algorithm race in Balsa-II while Figure 3(b) shows an image from the original BALSA with several view windows.

4.3. Zeus

The latest evolution of BALSA is Zeus (Brown, 1991), a system written for Modula-3 on DEC workstations. Like BALSA, Zeus supports multiple synchronized views and allows users to edit them. The user can change a data item's representation, once the animation is stopped, by text-editing, direct manipulation, or invoking a function in the typescript window. All of the other views are updated following the change, and if the program continues it does so with the altered data. Figure 4 shows a snapshot (and movie) of a Zeus session with several of the views. Zeus is also noteworthy for its object-oriented design, graphical specification of views, and the fact that it is implemented in a multi-threaded, multi-processor environment, so it can easily animate parallel programs. Brown has also had considerable success with audio enhancements to Zeus using a MIDI synthesizer; his work with Hershberger (1991) is an excellent introduction to the use of colour and sound in algorithm animations. The most recent work using Zeus has been a three-dimensional extension (Brown & Najork, 1993) which has examples illustrating uses of 3-D for encoding extra information. As it is a prototype, Zeus has seen limited use outside the laboratory and no empirical evaluation has been performed, but it has been used as a tool for algorithm development by a number of researchers.

4.4. TANGO

TANGO (Stasko, 1989; Stasko, 1990), also produced at Brown University, was built by Stasko on an algorithm animation framework which encompasses a new *path-transition paradigm* for easily incorporating smooth transitions into any algorithm animation. Most SV systems have used a simple erase-redraw technique for animating displays; TANGO allows visualizers to produce smooth cartoon-quality animation without the overhead of writing specific code for each step of a transition.

Stasko's framework is composed of three parts: defining the abstract operations and events in the program that drive the animation, designing the animation to simulate the abstractions and operations, and mapping the program's operations to the corresponding animation scenes. The first part involves inserting "algorithm operation" calls in the source code, which can be done manually or with a special editor which does not modify the code itself.

The animation design part of the framework centres on the path-transition paradigm which introduces four abstract data types for defining transitions. These can define a trajectory (using an algebra of paths) or changes in size, colour, or visibility. Stasko also built an "example-based programming" tool (Stasko, 1991) which can automatically produce the animation code from a user-demonstrated example of the animation.

The third component of the framework is the mapping from the actual program code and its execution to the animation. BALSA only allowed a one-to-one mapping between events in the program and the animation scenes, but Stasko uses a formal framework that supports one-to-many, many-to-one, and many-to-many mappings. Thus the same animation components can be repeatedly referenced by many different algorithm operations within one program.

To animate a program in TANGO, the visualizer would first annotate the C source code with *algorithm operation* calls. Next, the animation scenes are designed, which are written as C functions that receive information from the driving program. Visualizers can make use of macro packages to create logical structures of objects using the four abstract data types. The final step is to write the animation control file which defines how algorithm operations map to the animation scenes. The animation is then ready to be run and the user is provided with rudimentary navigation controls for panning and zooming as well as pause, continue, and speed control.

The original version of TANGO ran on BSD Unix workstations using a customized graphics package developed at Brown and provided silent two-dimensional colour animations of a wide variety of programs. The current version, called XTango (Stasko, 1992), runs on most popular Unix workstations that can compile C and use the X11 Window System. It differs from most SV systems in that it is widely and easily available to the general public along with a large library of animated algorithms. Figure 5 shows a still image (and movie) of the smooth animation.

4.5. ANIM

ANIM (Bentley & Kernighan, 1991a; 1991b; 1992) is a simple but powerful SV system for producing both animated visualizations on a workstation (or simple terminal) as well as static snapshots ready for inclusion in documents. Developed at AT&T Bell Laboratories, ANIM follows the Unix tool philosophy of having a simple generalized interface that is language and software independent.

ANIM is a "post mortem" system for making animations (`movie`) and still images (`stills`) from a script which is created as the program is run. Since the script is a set of plain text commands, ANIM is language independent; any program can be annotated to output the commands. With only eight commands in total (four drawing commands: `line`, `text`, `box`, and `circle`; and four control commands: `view`, `click`, `erase`, and `clear`), it is easy to learn. Visualizers can get a visualization from a program simply by inserting some print (or print-to-file) statements at interesting points in the source code, and then compiling and running it. When the program has finished running, the resulting output script can be fed to `stills` for images to be incorporated into a document, or to `movie` for an interactive animation.

The authors chose a post mortem technique rather than a "live" system (where the visualization is displayed while the program is running) because it allowed rapid development using established Unix tools across a variety of architectures. The script can be viewed interactively on terminals of varying quality and the same script can be used to produce still images for documents. A post-mortem system can know the display space range that will be used throughout the visualization and can scale everything appropriately. A dozen lines of `awk` code allow the synchronization of different simultaneous views by merging a number of ANIM scripts to show an "algorithm race". Figure 6a shows the `stills` version of an algorithm race produced with such a script while Figure 6b shows the `movie` version of a quicksort. Because the ANIM script language allows objects in the script to be named, it is easy to produce simple animations (since the re-drawing of an existing named object causes the old object to be erased). Since everything about the animation can be known in advance, it is possible for the user to have complete control of the animation speed and direction (forwards or backwards).

Although there has been no formal empirical evaluation, the authors report a number of practical applications from their users (ANIM is currently a limited release research prototype). In giving a short lecture on sorting (for which *Sorting Out Sorting* was too long and covered some irrelevant algorithms), one author used ANIM to produce a five minute videotape covering five sorting algorithms. The sorting algorithms took two hours to write the 74 lines of `awk` required; a further two hours were spent setting up and shooting the video. Applications from other users included: a 3-D molecular modeling system, a 3-D stereo viewer on an SGI Iris workstation, parse trees of lambda calculus, numerical analysis visualizations, displays of simultaneous differential equations, the debugging of a matrix manipulation program, computational geometry abstractions, parallel algorithm visualization, and dynamic statistical displays. The authors' conclusion is that a simple, easy to use, and widely available animation system with a generalized interface can satisfy the needs of teaching, research, and everyday computer programming.

4.6. Pascal Genie (includes Amethyst)

Conventional textual debuggers have a distinct advantage over most SV systems in that they can be used to easily examine the contents of a variable without the effort involved in constructing a visualization. Automatic SV addresses this point by providing visualizations of arbitrary programs with minimal effort on the part of the visualizer or user. Myers's Incense prototype (Myers, 1980; Myers, 1983), created at Xerox PARC, was the first SV system to automatically create graphical displays of program data structures. Myers and his colleagues at Carnegie-Mellon later created a production system called Amethyst (Myers, Chandhok, & Sareen, 1988) which was integrated with the MacGnome Pascal programming environment for Macintosh computers which is now known as Pascal Genie (Chandhok, *et al*., 1991).

Incense ran on Xerox's Alto computer and took advantage of its mouse and bit-mapped display (both innovations in commercial systems at the time). It was written for Xerox's Mesa language, a strongly-typed language similar to Pascal, and intended for use by experienced programmers. The default displays (composed of text, boxes, lines, splines, and arrows) included all of the basic Mesa data types as well as records, arrays, and pointers. The prototype read the symbol tables from the Mesa compiler and automatically produced a "natural" graphical abstraction of program variables. This abstraction came from a built-in default for the data type and could be customized. Incense provided a framework for programmers to define their own visualizations, but Myers speculated that creating the custom visualizations was often more difficult than the actual data manipulation algorithms, so programmers were not likely to create them (Myers, 1980).

Several of the basic ideas from Incense were incorporated into Amethyst, which can display both static and animated representations of data structures for Pascal programs written by the user. Amethyst differs from Incense in that it was specifically designed for use by students was built in an integrated environment which included facilities for automatic display management and animation. As with Incense, it provides default displays for each of the simple data types, although it uses a convention of showing each type in a characteristically shaped box. Aggregate types shown in nested boxes, visualization of pointers, and support for detail suppression have been implemented since the last report. Amethyst is integrated into the MacGnome structured programming environment which is sold commercially as the Pascal Genie. The Pascal Genie has been used in secondary schools and undergraduate courses for several years and is in use daily by hundreds of students around the world. Empirical evaluations (Goldenson, 1989) suggest that using the entire environment (including but not restricted to Amethyst) is more effective than conventional program editing. Figure 7 shows a static view from the Pascal Genie environment.

4.7. UWPI

The University of Washington Program Illustrator (UWPI) (Henry, Whaley, & Forstall, 1990a) went one step further than both Pascal Genie and the conventional textual debuggers: it automatically provided visualizations for the high-level *abstract* data structures designed by the programmer, as opposed to the *concrete* data structures which appear in the implemented code.

UWPI (Henry, Whaley, & Forstall, 1990b) was implemented using 14,000 lines of Allegro Common Lisp on a DECstation 3100 running the X11 windowing system. It was also made publicly available by ftp and we found it relatively easy to port it to Harlequin LispWorks on a Sun SparcStation. It could animate abstract data structures in programs written in a subset of Pascal which included block structure, structured control statements, constant declarations, integers, and one- or two-dimensional arrays. Users could simply feed the source code for their Pascal-ish program to UWPI which would analyze it and run the program through its interpreter. As the program ran, a pretty-printed version of the source code was shown with the current line highlighted. A data illustration window showed abstract views of the data structures according to the (previous) analysis. Users could manually pause and continue the program at any time by clicking on screen buttons, and could speed up or slow down the execution in 50 ms increments. The mouse-sensitive source code view allowed users to mark breakpoints in the code on variables (read or write) or statements (execute). Figure 8 shows a still from an animation of an abstraction for a breadth-first search.

The heart of UWPI was its "inferencer" which analyzed each of the data structures in the source code and suggested a number of plausible abstractions for each. Each of these was assigned a weight based on the closeness of the fit between the permitted operations on the abstraction and the operations found in the code. The abstraction with the highest final weight was chosen for each data structure in the code and the results were passed on to the "layout strategist". The layout strategist knew which graphical representation to use for each data abstraction and it chose the largest and most complex one to use as the backdrop while the program was running, with all other abstractions moving on top.

In the released prototype, the authors implemented abstract representations for booleans, pointers, indices, magnitude variables, enumerations, relations (or digraphs), queues, and general linked lists. These allowed for reasonably effective visualizations of array sorting and graph search algorithms. UWPI was not designed to provide any kind of screen management nor was it intended to deal with large programs. The authors did not provide tools for experimenting with new graphical abstractions and most of the visualizations were borrowed from other popular SV systems. No empirical evaluation of the system or its visualizations were reported and UWPI is not being developed further.

Although UWPI appears to have high level knowledge of the program in that it showed graphical abstractions of data structures without any help from the programmer, it does not have a deep understanding since it did not know what the program was doing; it only gathered shallow information about how data structures were used and matched against a rule base. The authors note that their knowledge acquisition method was informal in that they simply began with an approximation of a rule and fine-tuned it until it worked for a particular example. Nonetheless, UWPI is the best example to date of automatic SV with any kind of applied intelligence.

4.8. SEE

One of the easiest yet rarely used methods of automatic SV is the pretty-printing of source code. When source listings *are* pretty-printed they rarely make use of the wide typographic vocabulary available in even the most rudimentary word processing software (such as introductory textbooks which only show keywords in bold). The SEE visual compiler prototype (Baecker & Marcus, 1986; Baecker & Marcus, 1990) is a fully automatic and customizable SV system which makes extensive use of human factors research in typography to produce high quality hard copy listings of C programs.

The model used by SEE's designers was to think of a printed computer program listing as a book or technical manual, with a table of contents, chapters, and indices. Since SEE is a compile-time rather than run-time SV system, all of the information contained in the visualization is static and does not relate to any particular execution of the code. In a typical SEE listing of a C program (with the default parameters), functions appear in a large bold font with a thick line running across the page in the same way that a major section heading in the chapter of a book might appear. Parameters and local variables appear in a neat two column (type-variable) list under the function header and all C source code is printed in a sans-serif font. Multi-line comments appear in a serif font on a gray background while in-line comments appear in a smaller serif font in the margin beside the line that they annotate. Figure 9 shows a typical function with comments.

Several of the ASCII characters in the C source code were changed by SEE in the printed output. The braces `{}` used to denote scope were removed (with one exception) and replaced with systemic indentation to encode visual hierarchy. The `->` pointer dereference and record element retrieval was replaced by a simple arrow ’. Some of the more indistinguishable operators had their legibility enhanced by using superscripting or a larger font to make them clearer and each operator and keyword had its horizontal spacing individually adjusted to improve clarity.

In their example "Program Book", the authors presented several views of the source code for an implementation of Eliza (Weizenbaum, 1966) that was produced "almost totally automatically" (except for minor fix-ups relating to page breaks, headers, and footnotes). Two tables of contents were given: the first gave the "program meta text" showing user documentation, overviews of the code, profiles, the main module names, programmer documentation, and various indices to the code; the second was for the "program text" and showed each file, function, and the data objects it contained with a page reference for each. One of the code overviews included a "program map" which showed each page of code scaled down to postage-stamp size with a label in a readable font beside each function declaration, thus allowing the programmer to navigate by the "shape" of the functions. The book representation also provided a simple cross-reference in the footer of each page pointing to where global variables used on that page were originally defined.

Two other pretty-printing systems are the `vgrind` utility (BSD Unix Distribution, 1988), which is part of the Berkeley Unix distribution, and WEB (Knuth, 1984) which is part of the TEX text mark-up language distribution. The `vgrind` utility is language independent but it simply shows program keywords in bold, comments in italics, and function names in the margin. With WEB, Pascal programs and their documentation are integrated in one TEX document, which means that existing non-WEB programs cannot be visualized automatically and which requires programmers to edit marked-up code. This could be avoided by implementing a SEE-like pretty-printer in a WYSIWYG editor on a workstation. A prototype of this for the Turing language was developed by Small (Small, 1989), but we are surprised that this technology has not been exploited given the speed (and idleness) of conventional desktop workstations. SEE itself is no longer being developed.

The authors subjected SEE to a modest empirical evaluation which revealed some improvement in comprehension (compared with conventional plain listings) among novices for a 200 line program.

However, the same study also revealed a slight decrease in comprehension for a similar program which was much more "dense" and had a large number of embedded comments. Although the evaluation is ambiguous, we believe that clearer presentation of source code is certainly an important part of SV.

4.9. TPM

Although much of the visible SV work has dealt with imperative languages, one of the most successful automatic systems is used as a graphical tracer and debugger for the declarative language Prolog. The Transparent Prolog Machine (TPM) (Eisenstadt & Brayshaw, 1986; Eisenstadt & Brayshaw, 1988; Brayshaw & Eisenstadt, 1991), developed at the UK's Open University, was first announced in 1986 but has since undergone considerable development and is now available in both commercial and public domain versions. It uses an annotated tree with extensive navigation and trace facilities to explain the execution of both large and small Prolog programs to expert and novice programmers alike.

TPM itself is written in Prolog and, although the original version was developed on Apollo workstations, implementations now exist for a variety of 680xx-based and 80x86-based workstations (Eisenstadt, Brayshaw, & Payne, 1991; Payne, 1991) and the most recent version runs on the Macintosh (Kwakkel, 1991). TPM provides two basic views for the user: the coarse-grained view (CGV) and the fine-grained view (AORTA diagram). The CGV is based on a traditional and/or tree; it shows the execution space of the entire program with nodes representing goals. Each node is coloured to indicate the current state of the attempted goal (pending, succeeded, failed, or initially succeeded but failed on backtracking). The trees for non-trivial programs are usually too large to fit on one screen, in which case traditional window scrolling techniques must be used. TPM also provides a very small scale version of the diagram with a "you are here" marker for tree-wide navigation on large programs. Squares are used to represent user-defined code, whereas circles are used to depict system primitives. The user may also compress selected nodes to elide uninteresting subtrees, which are represented by triangles.

The fine-grained view allows the user to zoom in on a particular node to get details of data flow, such as variable instantiation. This view is called an AORTA diagram (for "And/OR Tree, Augmented"); it shows the and/or tree hierarchy for some subtree with each goal node represented by a box divided by a horizontal line with a number of small vertical lines hanging off the bottom to indicate the goal's clauses (clause branches). The top of the box indicates the current status of the goal (a question-mark indicates that it is pending, a check-mark indicates success, a cross indicates failure, and a crossed check-mark indicates initial success followed by failure on backtracking) while the bottom of the box contains the number of the clause currently being considered. The clause branches indicate the status of each clause in the current goal and if there were prior invocations of the goal then it is shown with a shadow. The current goal and its matching clause are shown in text beside each AORTA node with arrows showing input and output unification of variables and small "lozenges" to show variable instantiations that have come from elsewhere. Figure 10 shows an example of the CGV and basic control panel while Figure 11 shows an AORTA view of a single goal node from the classic definition of "append".

TPM can run in live or post-mortem mode, with the advantage of the latter being that the entire tree is known so it can be laid out optimally (in live mode the user can request a re-drawing at any time). The user interface provides standard "video replay panel" buttons for rewinding the trace to the beginning or end, playing, fast forwarding, or single stepping forward or reverse. Some degree of customization is provided to allow users to filter the goal tree and specify how a node and its links should appear.

In order to show that TPM is suitable for debugging large programs, the authors ran it on their own in-house knowledge engineering toolkit called MIKE (Eisenstadt & Brayshaw, 1990; Bessant, 1991), which consists of over 2200 lines of code. In their example debugging session (using post-mortem mode), the entire execution space of the program was shown and both CGV and AORTA views were used to successfully locate a bug, thus suggesting that TPM might be useful to experienced Prolog programmers working on large programs. TPM and the AORTA notation have also been used extensively by students in the Open University's distance teaching programme (Eisenstadt, 1988) as well as interactive classroom teaching. No formal empirical studies have been performed on TPM, but Open University examiners have found that students spontaneously use AORTA diagrams in their ungraded "scratch" work when writing their course exam, which suggests that they find the notation helpful.

4.10. Pavane

Traditional SV systems have dealt with sequential programming languages but recent work in high speed computing has concentrated on parallel and massively parallel architectures. Programming and debugging in a parallel language can be much more complex than in sequential programming because of the greater number of computational elements interacting and the non-deterministic way in which a parallel program can execute. Several researchers have noted that concurrent programming is an excellent domain for SV because of the large amount of highly dynamic information present in a concurrent computation. Researchers at Washington University (St. Louis) have developed a system called Pavane (Roman, *et al*., 1992b) which is designed to provide declarative three-dimensional visualizations of concurrent programs written in Swarm (Roman & Cunningham, 1990), with recent extensions allowing visualization of C programs.

Pavane was implemented in five parts: a parser which translates Swarm programs and their visualizations into an intermediate language (originally Prolog, but the most recent version uses C), a run-time package for the execution of compiled Swarm, a library of routines that is used when visualizing C programs, a second run-time package for the execution of compiled visualizations which produces the animation trace, and a viewing program which renders the visualization and provides the user interface.

All these components are implemented in C and run under Unix; the viewing program runs on an SGI Personal Iris. A visualization consists of an underlying computation (either Swarm or C), a visualization computation (compiled visualization rules), and a viewing computation, all of which can communicate over Ethernet. In principle, this division allows any kind of underlying computation (sequential or concurrent, from any architecture) to supply information about its state changes to the visualization computation. The viewing program allows the user to stop, continue, and step through the animation as well as rotate it in three-dimensions and zoom in and out.

Visualizations in Pavane are specified in a *declarative* style (Roman & Cox, 1989), in which the visualizer defines (declares) a transformation between the state of the computation and the final images. In the most simple case such a transformation may consist only of the declaration of a number of graphical objects whose parameters can be changed by program operations. But Pavane also provides much more powerful abstraction capabilities. In Pavane, the transformation is expressed as a series of mappings between spaces (collections of tuples), with each mapping composed of one or more rules. A rule specifies a relationship between two spaces--for example, "For every tuple value (i,j,v) in the rule's input space, there is a tuple box (corner = [i,j,0], xsize = 1, ysize = 1, zsize = v) in the output space".

The declarative style stands in contrast to the *imperative* style of the BALSA "interesting events" method where calls to visualization routines are inserted near "interesting" code. In an interpreted declarative system, if the computation’s interpreter is instrumented with "probes" then there is no need to modify the source code. This can also be done in compiled languages by having the compiler automatically insert calls in the code, but this does change the execution of the program even though the source code that the user and visualizer see is untouched. This issue can be important in a parallel program where the execution is non-deterministic since invasive visualization code can change the outcome of a computation.

In addition to using this declarative specification technique, Pavane’s authors also have a sound methodological foundation for choosing which visualizations they specify. They assume that the properties of a program which are necessary to verify its correctness are also properties which should be visualized and they often refer to the first stage of visualization as the *proof mapping*. This mapping isolates the components of interest and helps decide how the components will be abstracted.

In their solution to the *Diffusing Computations Problem*, the authors use one of the key invariants in the correctness proof to decide the basic layout of the visualization. This problem describes a number of active and inactive processes which communicate with each other and an external "environment". The invariant states that the parent information for active processes defines a directed tree rooted at the environment, so the authors chose to represent the graph-theoretic tree geometrically and use it as a basis for the visualization. Nodes represent processes and edges show connectivity. The tree has a given planar layout, but since the distance from each node to the root is important in the proof, this is encoded in the visualization as the depth from the viewer thus making the tree three-dimensional.

The remaining important elements in the proof mapping are whether or not a process is active or inactive, which is encoded as a colour change from green to red, and whether or not a message is being passed, which is encoded as a line growing from one node to another. The authors have published a videotape showing this animation (Roman, Cox, & Boemker, 1990) as well as a more recent videotape showing a wide variety of 3-D visualizations (Roman, *et al.*, 1992a). Figure 12 shows a still image from a graph searching algorithm on this tape.

Pavane is an active research prototype and although it has not undergone empirical evaluation, the visualizations presented are a compelling indication of the power of 3-D colour animations to aid in the understanding of both sequential and concurrent programs.

4.11. LogoMedia

Although several researchers have experimented with the use of sound to enhance the understanding of software, few have devoted serious effort to providing a usable interface for adding audio to a visualization. LogoMedia (DiGiano, 1992; DiGiano & Baecker, 1992; DiGiano, Owen, & Rosenthal, 1992) is a prototype Logo programming environment developed by DiGiano at the University of Toronto which allows programmers to associate non-speech audio with program events while the code is being developed. The interface is specifically designed for specifying visualization events with sound and it provides a number of "smart" defaults for monitoring variables or control flow.

LogoMedia is based on an earlier prototype, LogoMotion (Buchanan, 1988; Baecker & Buchanan, 1990), which provided extensions to the Logo language for non-invasively associating special animation procedures with program events. LogoMedia uses a similar technique, but the focus of the current implementation is visualization using sound (or *auralization* as some have called it). LogoMedia is implemented as three distinct components: the Logo interpreter written in the C programming language, the LogoMedia interface and audio components written in C++, and the visualization class libraries written in Logo. It runs on an Apple Macintosh computer connected to a MIDI synthesizer or sampler through the Apple MIDI Manager. Although the highest quality sounds can only be produced from an external MIDI device, rudimentary audio playback can be achieved using an internal MIDI sound generator which uses the Macintosh speaker.

LogoMedia uses *probes* to specify visualizations. Probes have an *action* which is the collection of external commands to execute, and a *trigger* which determines just when during execution the action takes place. LogoMedia supports two types of probes: control probes and data probes. LogoMedia programmers use a special editor to annotate their software by connecting control probes to lines of code which trigger sound commands when execution reaches their associated lines. Data probes can be associated with arbitrary Logo expressions, such that changes to the expressions trigger sound commands as well. Sound commands can turn on synthesized musical instruments, play back sound samples, or make adjustments to a sound's pitch or volume. Such audio probes cause subsequent runs of a program to generate and manipulate sounds which can aid in the comprehension and analysis of the program's behaviour.

Figure 13 shows the LogoMedia editor window. Control probes can be attached to a line of code by selecting the line in the editor window and then choosing a probe from a special set of menus that are part of every LogoMedia document. Data probes can also be connected to Logo expressions using a similar graphical interface.. The programmer enters an expression in the Probe Sheet window shown in Figure 14, then chooses a probe to be triggered whenever that expression changes. The four icons correspond to different types of feedback which probes can generate in response to an executing program. From left to right these are: the audio probe for making sounds based on program events, the graphic probe for assisting in animation, the text probe for printing the messages and the values of program data, and the generic probe for specifying arbitrary Logo commands. Generic probes can be used to specify complex audio visualizations through MIDI primitives added to the Logo language or to call sophisticated visualization procedures defined elsewhere. In the current implementation only the audio and generic probes are fully functioning.

DiGiano's evaluation of the effectiveness of sounds in portraying program behavior and the appropriateness of the LogoMedia interface involved three subjects in a seven hour observational study. Subjects were introduced to the LogoMedia audio probing techniques and then were asked to compose a new program and identify and correct four bugs in another program. After only two hours of training, all three subjects were able to use sound in various ways to locate problems in their code and verify corrections. Subjects were observed using the audio tools to test coverage, monitor the call stack, identify infinite loops, and listen to one aspect of program data while another was changing on the screen. It is clear that more research and evaluation are needed in the use of audio to communicate complex information, but these results suggest that sound can be a significant aid to comprehension.

4.12. CenterLine ObjectCenter (formerly Saber-C[++])

Commercial programming environments have been relatively slow to take advantage of the workstation interface enhancements of the 1980's. The traditional tools employed by programmers trying to understand software, aside from comments and documentation, are program listings and cross-references. Those requiring a deeper understanding might employ a source-level debugger, such as

Unix’s `dbx`. The original textual version of `dbx` has a command line interface with the ability to list sections of source code, single or multiple step through a running program, set breakpoints, and print the value of simple variables. By the late 1980’s this kind of functionality had been dressed up with a conventional window interface where buttons could be used instead of line commands and separate windows could be devoted to showing the current source code and the values of any variables of interest. Many of the conventional programming environments in use today, such as Borland’s Turbo C on IBM PCs or SunPro’s SPARCworks on the Sun SPARCstation, use the same kind of variation on `dbx`. The ObjectCenter C and C++ (formerly Saber-C++) programming environment (CenterLine Software, 1991) extends `dbx`’s functionality slightly and makes some use of graphics to convey information and thus qualifies as a kind of SV.

Unlike conventional debuggers and tracers which really only provide run-time information about programs, ObjectCenter can also provide static compile-time information about the source code at any time. Using the source code browsing window (see Figure 15), which doubles as a basic mouse-based editor, users can select fragments of source code and invoke a menu command to get a pop up window showing all of the functions that could be called if that code were to be executed. Similarly, the definition of any symbol can also be shown in a pop up window. When a variable or function is selected a cross-reference browser can be invoked to show a graph of the selected object indicating all the places that the object is referenced and all of the functions and global variables that it references. Figure 16 shows the cross-reference graph for the function `bounce` in the class `DrawableShape`. The user can then select any of the objects in this graph and show a new cross-reference with the selected object as the focus. A similar interface is provided for browsing the class hierarchy showing base and derived classes as well as member functions and data members. ObjectCenter’s run-time environment has a distinct advantage over conventional debuggers in that it provides a C++ interpreter built in, much like the Lisp Listener. This allows the user to modify global variables or call functions at any time by simply typing commands and having them "executed" immediately, thus allowing for more experimentation without the overhead of re-compiling, linking, and loading.

The run-time environment provides all of the breakpointing, stepping, and display functionality of `dbx` as well as the ability to trace the currently executing line in the source code window, much like BALSA did (although it only shows the currently executing line as opposed to the stack of pending returned calls). As with the other conventional debugging tools, ObjectCenter can only display the basic C data types (int, char, float, and pointers) in its data views although it does display structs (records) with their programmer-defined field names. As with the compile-time information, when the running program is stopped at a breakpoint users can select variables and get a pop up window showing their current value (which they can edit if they wish). One distinct advantage of ObjectCenter over its conventional counterparts is its ability to deal with pointers. Conventional debuggers usually show a hexadecimal number (representing a location in memory) when asked to print a variable representing a pointer but ObjectCenter displays a small button beside each pointer in a data display. If the user clicks on the button then a line connects the pointer to a new window showing the contents of the object pointed to. Figure 17 shows a simple linked list which ObjectCenter has displayed by recursively following the pointer to the end of the list.

All of the information provided by ObjectCenter about the user’s program can be obtained by the older text-based tools. Cross-referencers, tag programs, and editors have been able to supply this kind of information in printed lists for many years, but the ability to see a cross reference graph and to quickly look up or follow function calls and variables at a mouse click (as opposed to typing commands and loading new files) represents a tremendous speed up in information retrieval and a decrease on the cognitive load of the programmer. Even though the simple data displays pale in comparison to those in Pascal Genie, the ability to quickly browse the data space and follow pointers at the press of a button gives this kind of visualization a distinct advantage over conventional methods.

5. Deriving the Basic Categories

Previous taxonomies of SV have not provided a principled basis for their choices of categories, instead relying on common characteristics observed and deemed by their authors to be important. This is a common beginning for a taxonomy in any field, but as SV becomes more mature and a wider variety of systems is implemented, it is clear that a more systematic approach should be attempted. In this section we describe the method that we used to derive the top level categories in our hierarchical taxonomy, while in the next section we fill out the hierarchy beneath each category in an attempt to describe as completely as we can the characteristics of systems observed to date.

Any piece of computer software can be modeled as a black box which produces some output data based on the particular input data supplied to it. Typically, the output feeds back through the user, who considers it and bases the next set of input upon it, as shown in Figure 18. The input data can be multimodal: it can consist of files, typed commands, gestures, sound or speech input. The input data can be temporally static or dynamic in nature. The black box encloses a set of arbitrarily complex algorithms, which for compiled languages are represented as machine-level instructions running on a specific hardware architecture. Conceptually, however, the black box consists of the high-level algorithms and data structures originally conceived by the programmer. The output data can also be multimodal and temporally dynamic. The user can interact with the input and output data infrequently, as is the case with batch processing of scientific data, or on a continuous basis, as is the case with highly interactive drawing applications.

This simple but complete model provides a conceptual basis for defining a structure for characterizing SV systems, since the SV software itself (and the software it is visualizing) can be described by this model. As there are four parties providing input to any software visualization (the programmer, SV system developer, visualizer, and user), we refine the model to show how each party provides a distinct part of the input to a visualization. Consider a particular piece of software, ***a***, which is visualized by a given software visualization system, ***b***. As shown in Figure 19, ***a***, its input and its output, comprise only one part, albeit an important one, of the input data to the SV system ***b***. The SV system will only be able to visualize certain aspects of the program and its I/O data, hence the second box below the program. Another key component of the input data is the actual specification of the visualization to be performed by ***b*** on ***a***. Finally, it is reasonable to expect that there may be some interaction with the user during the visualization process. These four components, representing the input from all parties involved, together characterize the complete input dataset to ***b***.

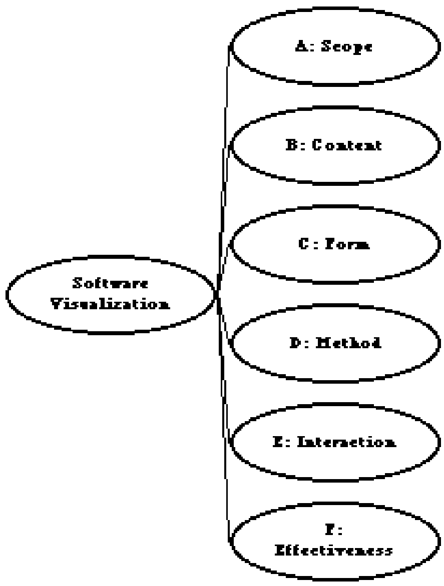
This analysis suggests the top-level categories of our taxonomy, which appear in Figure 19 as vertical labels. The first category, *A: Scope*, describes the range of software that can be handled by a given SV system. Few systems (if any) are capable of visualizing a completely unrestricted set of programs, and this first category sets the basic limits of a SV system. Even though the complete information about a piece of software may be available, no known visualization system actually uses all of it. Our second category, *B: Content*, describes the subset of information from *Scope* that ***b*** really uses in constructing the visualization. The most important element from the point of view of the user is the output of the visualization, which is described in category *C: Form*. Since it is difficult to capture the essence of multimodal output in a structured taxonomy, this category concentrates on specifying the parameters and limitations which govern the output. The style in which a visualization is specified is an important distinction between systems since it affects their usability from the perspective of the visualizer. Category *D: Method* characterizes the important elements of the visualization specification. Category *E: Interaction* characterizes a system's interactivity by analyzing the most important controls which are suitable for interactive manipulation.

Finally, while the SV system designer had a particular set of objectives in mind, there is no guarantee that the resulting system actually fulfills those objectives. Although our model does not indicate the designer's objectives nor how they affected the user, we believe that this is an important issue, otherwise the building of SV systems would have to be described as a black art with no measurement of a system's success. In order to evaluate the system's effectiveness objective empirical evidence is required and we propose a final category, *F: Effectiveness,* to provide a framework for such evidence.

6. Taxonomy Detail

Our derivation of the SV taxonomy yielded six distinct *categories,* as shown in Figure 20. We display these categories as a tree because they form the basis for our taxonomy, which, while extensive, is by no means "complete". As we mentioned earlier, a good taxonomy must be expandable to permit new discoveries to be catalogued and more detailed study in specific areas. We have identified a number of minor categories which make up each of the six major categories, and each of the minor categories may in turn have sub-categories which may in turn have sub-categories, and so on. Thus the entire taxonomy may be described by a multi-level n-ary tree. We have designed the taxonomy structure so that new categories and sub-categories may be added naturally without redesigning the entire tree.

Figure 20



Each category or sub-category that we use can be qualified for a particular SV system by a binary description (e.g. does the system support concurrent programs? yes/no), a range (e.g. to what degree does the system visualize data structures?), or a set of *attributes* (e.g. a subset of Pascal programs run under SunOS 4.2 on a Sun SparcStation running X11R5). For categories involving a range, we subjectively assign a ranking of lowest, below average, average, above average, or highest (see Table 7). These rankings are derived from our understanding of a system based on the papers written about it and in most cases by our own use of the system or by consultation with its authors. Hand-designed visualization systems like BALSA/Zeus, TANGO, ANIM, Pavane, and LogoMedia were difficult to rank in many display and interaction related categories because their output is highly dependent on what the visualizer has designed for a particular program. For these systems, we have ranked them using their known capabilities as demonstrated by the example visualizations that we have seen. While it may be possible to achieve a certain effect with a particular system, we only consider it a feature if the system supports it in some way. If it requires a great deal of programming or workarounds to achieve then we would rank it low for that characteristic. When a characteristic was completely dependent on the choices of the visualizer we gave a rank of average.

Relative rankings are based on our understanding of the current state of SV technology. Although these assignments are somewhat arbitrary, we believe that they are useful for rough relative comparisons between the abilities of various systems, and we understand that an absolute ranking may change as technology improves (e.g. a system which is ranked high in Intelligence today may be reassessed as medium in five years if AI technology improves markedly). A more formal ranking system is necessary for a fully developed taxonomy, but this taxonomy should serve as a guide to areas of research in need of development. In the sections below we define each major category and its associated minor categories and sub-categories and provide a relative ranking for each in Tables 1 through 6.

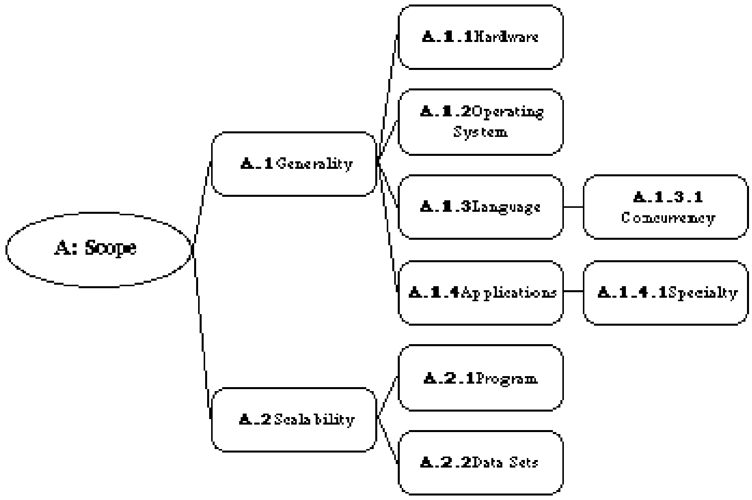
All of these categories and sub-categories *together* describe each SV system and the relative rankings indicate the strengths of each system in each of the areas. Because we see each system as a hybrid of many properties, it is not possible to "pigeonhole" a system using this taxonomy (as, for example, is done in the taxonomy of living things where each creature has followed a distinct evolutionary path with a low degree of cross-mutation).

A: Scope

What is the range of programs that the SV system may take as input for visualization?

We see two major divisions of Scope information: *generality* and *scalability*. In Figure 21 we show a complete tree of all of the sub-categories, described below, that we believe characterize the scope of an SV system. Table 1 summarizes the relative rankings for our twelve example systems against the Scope sub-categories.

Figure 21



A.1 Generality

Can the system handle a generalized range of programs or does it display a fixed set of examples?

A generalized system can generate visualizations of arbitrary programs within a particular class while an example system displays a (possibly flexible) visualization of a particular algorithm, system, or set of existing programs. SOS is an example, since its presentation is fixed on videotape, while the remaining systems are generalized to some degree. A generalized system will usually have some restrictions governing its capabilities:

*** A.1.1 Hardware: What hardware does it run on?**
*** A.1.2 Operating System: What operating system is required to run it?**
*** A.1.3 Language: What programming language must user programs be written in?**

*** A.1.3.1 Concurrency: If the programming language is capable of concurrency, can the SV system visualize the concurrent aspects?**

*** A.1.4 Applications: What are the restrictions on the kinds of user programs that can be visualized?**

*** A.1.4.1 Specialty: What kinds of programs is it particularly good at visualizing (as opposed to simply capable of visualizing)?**

The *hardware* platforms used by our example systems are split between Unix workstations and Macintoshes, probably due to the early availability of window-based graphics on these systems. Most of the systems, TPM, Pavane, and LogoMedia excepted, only work on traditional imperative languages, while ANIM is noteworthy for its ability to work in any language. In other languages, Lieberman (1984; 1989) has produced interesting systems for visualizing Lisp while London and Duisberg (1985) did pioneering work with Smalltalk. Of the five systems that visualize languages with *concurrency*, only Pavane is truly designed to produce visualizations of concurrent elements. We have also built a prototype for a procedural language which shows process activity on a static representation of the module and procedure hierarchy as well as individual views showing the status of each process (Price, 1990; Price & Baecker, 1991). The MRE system of Brayshaw (1990; 1991) can visualize the logic language PARLOG, a parallel version of Prolog. For a detailed overview of Concurrent SV systems, see Kraemer and Stasko's (1993) survey.

Although most of the example systems are technically capable of producing visualizations of any *application* in the appropriate language, most have a particular *specialty* outside which the visualizations are not very informative. SOS specializes in exactly nine specific algorithms while BALSA demonstrates some excellent visualizations of array and graph algorithms. Both Genie and ObjectCenter can show linked lists and trees well in their data views while UWPI can only visualize simple data structures for a small set of graph searching and array sorting algorithms.

A.2 Scalability

To what degree does the system scale up to handle large examples?

Scalability includes a combination of:

*** A.2.1 Program: What is the largest program it can handle?**
*** A.2.2 Data Sets: What is the largest input data set it can handle?**

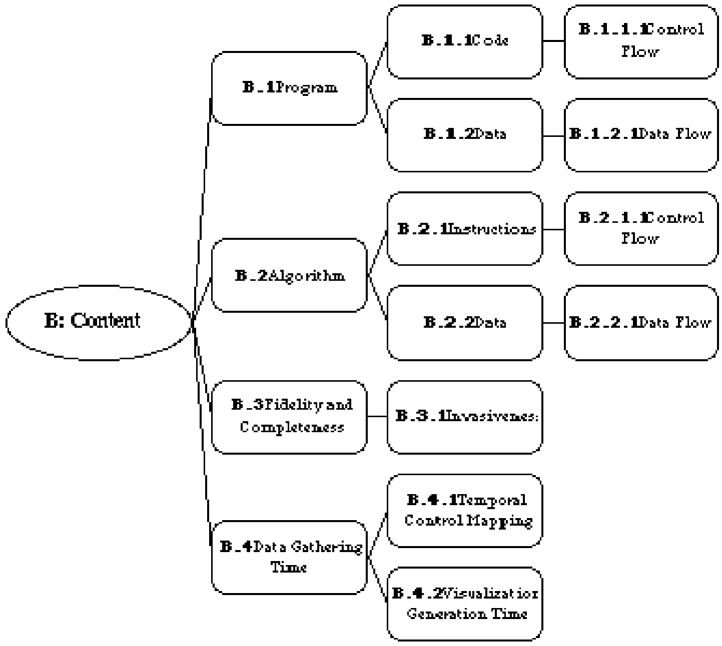
This characteristic refers to fundamental limitations of the system only; see category F: Effectiveness to determine how *well* it presents visualizations of large programs. Most of the systems are technically capable of visualizing large programs and data sets, but only TPM and ObjectCenter have demonstrated examples. SOS visualizes large data sets, albeit fixed, while UWPI and LogoMedia, as prototypes, would not be expected to work on large programs.

B: Content

What subset of information about the software is visualized by the SV system?

Two of the most important parts of this information are the *program*, by which we mean the program source code, and the *algorithm* or "high-level" description of the software. The differentiation between program and algorithm is subtle and can best be described from a user perspective: if the system is designed to educate the user about a general algorithm, it falls into the class of *algorithm visualization*. If, however, the system is teaching the user about one particular implementation of an algorithm, it is more likely *program visualization.* Signs that the line from algorithm visualization to program visualization has been crossed include displays of program code listings as opposed to higher-level abstract code diagrams, and labeled displays of the values of particular variables, as opposed to generic data displays. Some systems are sufficiently flexible to produce both types of visualization, depending on what the user desires and specifies. Note that many authors may refer to their system as "algorithm animation" when their visualization refers to program code or variables: this kind of system would be program, not algorithm, visualization in our taxonomy. The other two important parts of this category are *fidelity and completeness,* which characterize the accuracy of the visualization, and *data gathering time,* which describes the point at which information about the software is gathered. Figure 22 shows the complete tree for the sub-categories described below while Table 2 summarizes the relative rankings of the example systems.

Figure 22



B.1 Program

To what degree does the system visualize the actual implemented program?

This category is further subdivided as follows:

* B.1.1 Code: To what degree does the system visualize the instructions in the program source code?

* B.1.1.1 Control Flow: To what degree does the system visualize the flow of control in the program source code?

* B.1.2 Data: To what degree does the system visualize the data structures in the program source code?

* B.1.2.1 Data Flow: To what degree does the system visualize the flow of data in the program source code?

While the distinctions between *control flow* and *data flow* become blurred when considering languages or architectures that use a message-passing paradigm, these characteristics are still generally applicable. Examples of *code* visualization include pretty-printed source code, structured diagrams, and call trees. While the nature of the underlying code may be implicitly visualized by the way in which data evolves, this is not considered to be code visualization; a more concrete visualization of the code (either statically or in execution) is required. Program *data* visualization is characterized by drawings of compound data structures showing their contents in terms of simple data structures, while program data flow can be represented by data flow diagrams or live views of the call stack.

Table 2 shows that the debugging-type tools like Pascal Genie, SEE, TPM, LogoMedia, and ObjectCenter fall firmly on the program visualization side with SEE being notable for its code visualization and Pascal Genie for its data visualization.

B.2 Algorithm

To what degree does the system visualize the high level algorithm behind the software?

As with the program category, this can be further divided as:

* B.2.1 Instructions: To what degree does the system visualize the instructions in the algorithm?

* B.2.1.1 Control Flow: To what degree does the system visualize the flow of control of the algorithm instructions?

* B.2.2 Data: To what degree does the system visualize the high level data structures in the algorithm?

* B.2.2.1 Data Flow: To what degree does the system visualize the flow of data in the algorithm?

As Table 2 shows, the higher-level display systems like SOS, BALSA, Zeus, TANGO, UWPI, and Pavane are all firmly established on the algorithm side, although UWPI gets a lower ranking because its examples are not as closely hand-tuned as the others.

B.3 Fidelity and Completeness

Do the visual metaphors present the true and complete behaviour (Eisenstadt, et al., 1990) of the underlying virtual machine?

Systems designed for software engineering may pose stronger demands than do pedagogical systems, since the latter may wish to take liberties in order to provide simpler, easier-to-understand visual explanations. Automatic systems like UWPI may produce a misleading abstraction for a data structure while visualizers using a system like TANGO may only animate a particular part of an algorithm for expository purposes. As Table 2 shows, the highest fidelity and completeness values go to the systems that are tied closest to the program code while the hand-designed systems were difficult to rank because they depend so much on the individual visualizer.

* B.3.1 Invasiveness: If the system can be used to visualize concurrent applications, does its use disrupt the execution sequence of the program?

Disruptive behaviour is not desirable in a visualization system for concurrent applications, as the effect of activating the visualization system may change the relative execution rates of processes, thereby producing a different result. The system of Flinn and Cowan (Flinn & Cowan, 1990) used a bus monitor to avoid this, but all four of our example systems that handle concurrency are invasive.

B.4 Data Gathering Time

Is the data on which the visualization depends gathered at compile-time, at run-time, or both?

In general, systems which depend on data gathered solely at compile-time (such as SEE) are limited to visualizing the program code and its data structures. These systems cannot produce any visualization of the actual data values, since they do not have access to that (run-time) information. Visualizations of data gathered at compile-time are generally not animated, as there is no relevant temporal axis along which to change the visualization. Visualizations generated from data gathered at run-time can produce complex displays of the variable space used by the program, and often rely on animation for an intuitive mapping between the temporal aspects of the program in execution and the presentation of the visualization. This is the most common style in interactive visualization systems. UWPI, Pascal Genie, and ObjectCenter gather data at both compile-time and run-time to provide their data displays.

This category can be subdivided into two other temporally-related sub-categories which apply if the visualization is based on data gathered at *run-time*:

*** B.4.1 Temporal Control Mapping: What is the mapping between "program time" and "visualization time"?**
*** B.4.2 Visualization Generation Time: Is the visualization produced as a batch job (post-mortem) from data recorded during a previous run, or is it produced live as the program executes?**

If the visualization is based on information gathered at a single point in time during the program's execution, and generates a static visualization, then its *temporal control mapping* is "static to static"; the system generates a *snapshot* (Incense does this to draw data structure diagrams). If the visualization generated is animated, the mapping is "static to dynamic"; we do not know of any examples of such systems. If the visualization gathers information over a span of time during program execution, and produces a single still visualization based on that information, the mapping is "dynamic to static": the visualization system is generating a *trace* (the `stills` program from ANIM does this). If the visualization uses information gathered over a period of time during the program's execution to generate an *animation* (this is the most common type), then the mapping is "dynamic to dynamic".

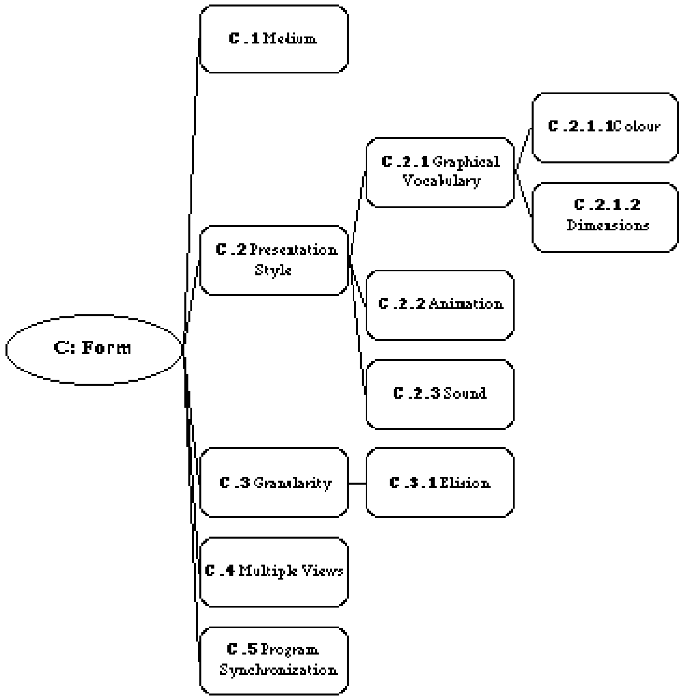
The *visualization generation time* affects how the user can interact with the visualization. The *post mortem* style of visualization (used by both ANIM and TPM) combines the advantage of the rich information available at run-time with the opportunity to lay it out in the most optimal way, since the entire range of display use (i.e. the "future") can be known in advance. This also has the disadvantage of the user being unable to interact with the visualization based on program output and thus have an immediate effect on the visualization. *Live* is the most popular method, although TPM has versions with both live and post-mortem modes. Live visualizations have the advantage of allowing the user to interactively specify the dataset, perhaps using a graphical tool to specify data abstractions such as trees or graphs.

C: Form

What are the characteristics of the output of the system (the visualization)?

This category is concerned with how the fundamental characteristics of the system are directly related to what can be displayed, which we have divided into five broad areas: *medium, presentation style, granularity, multiple views,* and *program synchronization*. Figure 23 shows the complete hierarchy while the rankings for each system may be found in Table 3.

Figure 23



C.1 Medium

What is the primary target medium for the visualization system?

While systems which are designed for one medium can often run on another (e.g. SEE, which was designed for a paper medium, can easily produce visualizations on workstations which support Display PostScript), we only list the primary target medium. Common choices include paper, film or videotape, plain terminal, or graphical workstation. We expect virtual reality environments eventually to become a target medium of SV systems. The most common medium for the systems in Table 3 is a black & white workstation monitor, with Zeus and Pavane being notable exceptions for their use of colour. SEE is an exception in that it only uses paper (although it could easily be implemented on a workstation monitor) while ANIM is capable of producing paper output using the `stills` program.

C.2 Presentation Style

What is the general appearance of the visualization?

Presentation style has no individual rankings because it simply serves to group the following sub-categories:

* C.2.1 Graphical Vocabulary: What graphical elements are used in the visualization produced by the system?

*** C.2.1.1 Colour: To what degree does the system make use of colour in its visualizations?**
*** C.2.1.2 Dimensions: To what degree are extra dimensions used in the visualization?**

* C.2.2 Animation: If the system gathers run-time data, to what degree does the resulting

visualization use animation?
*** C.2.3 Sound: To what degree does the system make use of sound to convey information?**

A system's *graphical vocabulary* provides some idea of the complexity of the visual primitives which make up the system's displays. Bertin (1983) describes the primary feature of a graphical vocabulary as being made up of the *marks* used (which could be individual *point* objects, *lines*, or *enclosed shapes*) which have a *position* in space. Each mark has six orthogonal retinal subtypes: *colour, size, shape, gray level, orientation,* and *texture*. All of these can be used to encode information and an SV system can be characterized by the size of its graphical vocabulary.

Colour can be used to convey a great deal of information while imposing a low cognitive load but has been greatly under-utilized in SV systems. Brown and Hershberger (1991) note five effective uses of colour: to reveal an algorithm's state, to unite multiple views, to highlight areas of interest, to emphasize patterns, and to capture history. As the table shows, relatively few systems have made extensive use of colour in visualizations.

Traditional systems have used simple two-dimensional graphics, although some recent work has used projections of 3-D images onto a 2-D screen. Pavane does this and provides tools for rotating the image in 3-space. Some high speed workstations now allow alternating polarized spectacles to be synchronized with the screen display to provide a binocular depth illusion. Some displays in ANIM have been constructed using this 3-D technique. The emerging virtual reality systems promise to provide an even better 3-D display and techniques such as Stasko's 3-D graph traversal (1992) may become commonplace. Note that simply *using* a 3-D display to show information that is naturally three-dimensional is not the most effective use of the extra dimension; Brown and Najork's Zeus-3D system (Brown, 1992), Stasko's sorting animation in POLKA-3D (1992), and recent work with Pavane (Roman *et al*., 1992a) are examples of the use of the extra dimension to encode non-dimensional information. The 3-D views in ANIM are simply a 3-D projection of naturally 3-D data.

The most obvious and frequent use of *animation* in program visualization systems is to capture and convey the temporal aspects of the software in execution. Does the system make use of animation in any other novel ways? Note that animation is not a binary characteristic; rudimentary erase-redraw techniques such as those found in UWPI are considered to be animation when compared with purely static visualizations such as SEE, but they compare poorly with the smooth animations found in TANGO or Pavane.

The audio output capability of most computers in the early 1980's was limited to a single beep, but most modern workstations have digital *sound* capability. Gaver and his colleagues (Mountford & Gaver, 1990; Gaver, O'Shea, & Smith, 1991) have demonstrated how sound can be used to communicate complex information, but of the systems we are considering, only LogoMedia and Zeus have made effective use of it. Brown and Hershberger (Brown & Hershberger, 1991) have identified four distinct uses of sound in SV: to reinforce visual views, to convey patterns, to replace visual views (so that visual attention may be focused elsewhere), and to signal exceptions. Other work includes that of Francioni, Jackson, and Albright (1992), who have investigated the use of sound in parallel programs.

C.3 Granularity

To what degree does the system present coarse-granularity details?

Many systems can visualize fine-grained details in a manner similar to that of a debugger, but the ability to filter out fine-detail to get the big picture can be an advantage. The "table of contents" view in SEE or the CGV in TPM are examples of built-in coarse-grain visualization support. A sub-category of granularity is:

*** C.3.1 Elision: To what degree does the system provide facilities for eliding information?**

An important feature for dealing with large amounts of information at one level of granularity is the ability to *elide* or temporarily hide sections that are not of immediate interest. As the table shows, the only systems to provide this feature to any degree are Pascal Genie, SEE, and TPM.

C.4 Multiple Views

To what degree can the system provide multiple synchronized views of different parts of the software being visualized?

Multiple views might include simultaneous coarse-grained and fine-grained views of data structures, or a graphical view of changing program data with a corresponding view of the executing source code. This is different from the *program synchronization* characteristic because it shows different synchronized views of the *same* program as opposed to a race between different programs. Zeus has the best facility for multiple views because it allows each view to be edited by direct manipulation with the result directly affecting the data. ANIM can also produce multiple views although they require some work on the part of the visualizer. Pascal Genie, SEE, TPM, LogoMedia, and ObjectCenter all provide multiple default views.

C.5 Program Synchronization

Can the system generate synchronized visualizations of multiple programs simultaneously?

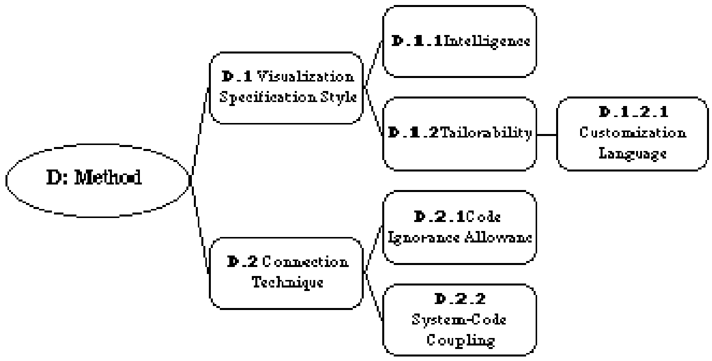
This capability is useful for comparing the execution speeds of two programs (by running a race), for determining how one algorithm differs from another similar algorithm, and for investigating how a particular algorithm is flawed with respect to a correct algorithm. Note that modern windowing systems and operating systems will allow almost any window-based visualization system to be run in parallel. Running two versions of a system on two algorithms in different windows would not qualify under this category because there is no centralized control or synchronization between both running visualizations. Of the example systems, only SOS, BALSA, Zeus, and ANIM provide this feature.

D: Method

How is the visualization specified?

This area describes the fundamental features of the SV system which the visualizer uses to create a visualization. We have divided this into two areas, one describing the *style* in which the visualizer specifies the visualization and one describing the way in which the visualization and the program source code are *connected*. Figure 24 shows the complete set of *method* categories and sub-categories while Table 4 indicates the relative rankings of the example systems.

Figure 24



D.1 Visualization Specification Style

What style of visualization specification is used?

Visualizations can be completely *hand-coded* (the user writes special purpose programs which visualize a particular algorithm or program) as with ANIM, or they can be built from a *library* or hierarchy of existing visualizations as with BALSA and Zeus. SOS is a *fixed* system since it cannot be changed from the original specification (the programmer, SV system builder, and visualizer were all the same person). Debugging tools (Pascal Genie, UWPI, SEE, TPM, and ObjectCenter) which are tied tightly to the program code all use some degree of *automation* to specify the visualization, thus making them appropriate tools for programmers. The remaining systems use some degree of hand coding to produce visualizations, with BALSA/Zeus and TANGO providing a *library* from which to build them. Even tools which use a library can be refined further: TANGO provides specialized tools for describing smooth trajectories and the ALLADIN system of Heltulla et al. (1989) provides rich graphical editing tools, thus making a high level specification of the visualization easier. Automatic systems have the advantage of making the programmer, visualizer, and user into the same person. They also take much of the work out of the visualization step and are more likely to be used by a professional or novice programmer looking for debugging assistance (Eisenstadt, Price, & Domingue, 1993). The sub-categories of visualization specification style include:

* D.1.1 Intelligence: If the visualization is automatic, how advanced is the visualization software from an AI point of view?
* D.1.2 Tailorability: To what degree can the user customize the visualization?

* D.1.2.1 Customization Language: If the visualization is customizable, how can the visualization be specified?

Automatic visualizations using syntactic information to generate hierarchy diagrams, such as TPM, use a "low" degree of *intelligence* while systems that are able to recognize algorithms or high-level data structures and display abstractions of them would be classified as "higher" intelligence. UWPI uses a moderate degree of intelligence because it can display abstractions, even though it does not "understand" them. VCC (Baeza-Yates, Jara, & Quezada, 1992) is an automatic prototype system which similarly generates abstractions of data structures in C programs and we are also working on more intelligent SV tools for C (Eisenstadt, Price, & Domingue, 1993). As Table 4 indicates, intelligence is sorely lacking among automatic SV systems.

In terms of *tailorability*, SOS is *fixed* and cannot be customized at all, while BALSA allows some interactive manipulation on the part of the user. The current X11 version of TANGO only allows window resizing, scrolling, and zooming but the original also had an interactive editor for customization. LogoMedia allows the user to choose which parts of the program will be visualized and which sounds will be associated with them (as well as change them while executing), while Zeus allows representations of data to be changed live by direct manipulation with the corresponding data in the program and the other views changed automatically. Note that being able to visualize different data sets does not qualify as customizing the visualization; the layout or presentation of the visualization must be changeable by explicit user instruction.

The *customization language* describes the way in which the system can accept customization instructions from the user. Systems which support *interactive manipulation* of the visualization, such as Zeus, have their visualizations specified interactively through direct manipulation. Systems which require the user to program explicit visualization code, like ANIM, rely on *procedural visualizations*. Systems which allow the user to describe the desired visualization using high-level tools or declarations, such as Pavane, support *declarative specification*. SEE uses simple command line flags to customize the printouts. Visualization systems can easily support more than one of these approaches for different aspects of the complete visualization specification.

D.2 Connection Technique

How is the connection made between the visualization and the actual software being visualized?

Systems like ANIM and BALSA require the visualizer to *instrument* their code by adding statements to print visualization commands at interesting events. TANGO, with the Field Environment (Reiss, 1990), and LogoMedia allow the visualizer to *annotate* a user program using a special editor so that the original source code remains unchanged (although the executable code differs from the unannotated version). It is also possible to have code *automatically annotated* by a pre-processor before it is compiled or interpreted, but all of these techniques are an *invasive* form of connection and may be dangerous from a software engineering point of view. Pavane, LogoMedia, and TPM provide an instrumented execution environment (interpreter or compiler) which allows the visualizer to attach non-invasive *probes* to data structures or code in a declarative manner so the structures can be monitored without affecting the source code at all. Another non-invasive technique is the use of a monitor which "listens" to the bus or another part of the hardware and shows a live report of commands, such as the work of Zimmermann et al. (1988). Note that one system can use several connection techniques. Other sub-categories related to connection technique include:

*** D.2.1 Code Ignorance Allowance: If the visualization system is not completely automatic, how much knowledge of the program code is required for a visualization to be produced?**
*** D.2.2 System-Code Coupling: How tightly is the visualization system coupled with the code?**

As one might expect, all of the automatic systems have a high *code ignorance allowance* since they can produce visualizations without any code knowledge on the part of the visualizer, which is one of the main attractions of this approach. Visualization systems which require modifications to the program source, however, require the user to "know" the program in order to produce a visualization of it. Systems which provide hooks or probes to which users can attach visualization code may require some knowledge of the program if the user wishes to make the best use of the potential probes available.

System-code coupling is a measure of how closely the SV system is tied to the program it is visualizing.

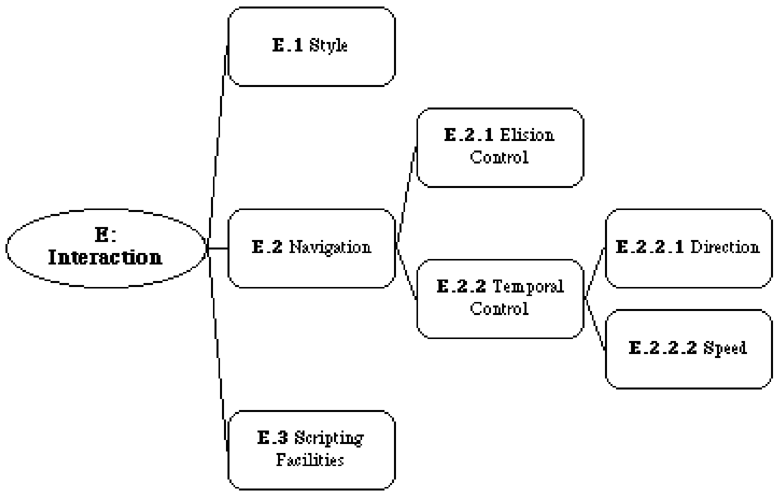
Systems like BALSA are tightly coupled and require programs to be run and visualized within an environment while some post mortem systems like ANIM do not require any coupling at all since the visualization system simply reads an output file produced by print statements in the program. Most of the systems are tightly coupled with the programs that they are visualizing, with TANGO, SEE, ANIM, and ObjectCenter being the exceptions.

E: Interaction

How does the user of the SV system interact with and control it?

We have found three major areas where interaction issues affect the fundamental design of SV systems: *style*, *navigation*, and *scripting facilities*. The complete hierarchy is illustrated in Figure 25 and Table 5 shows how the example systems are ranked according to the following categories.

Figure 25



E.1 Style

What method does the user employ to give instructions to the system?

Examples include on-screen buttons, menus, command line statements, or scripted programs. A variety of techniques may be employed by a system depending on the action required. As the table shows, the most common style of interaction involves buttons and menus (as with most window-based programs).

E.2 Navigation

To what degree does the system support navigation through a visualization?

This is especially important when considering very large programs or datasets. If the system and its navigation tools do not scale up then they will not be useful for professional programmers. Eisenstadt et al. (1990) suggest that navigability may be achieved by changes of resolution, scale, compression, selectivity, and abstraction. As the table shows, few systems other than SEE and TPM support large space navigation. Other navigation-related sub-categories include:

*** E.2.1 Elision Control: Can the user elide information or suppress detail from the display?**
*** E.2.2 Temporal Control: To what degree does the system allow the user to control the temporal aspects of the execution of the program?**

*** E.2.2.1 Direction: To what degree can the user reverse the temporal direction of the visualization?**
*** E.2.2.2 Speed: To what degree can the user control the speed of execution?**

Elision control techniques are useful for information culling, the removal of excess information which is not relevant to the user's line of inquiry and which serves only to clutter the display. This also applies to audio visualizations, since temporal elision (speeding up sounds) can suppress audio detail. Elision is of primary use with large problems, for which the entire dataset cannot be simultaneously displayed. As the table shows, few systems other than Pascal Genie, SEE, and TPM provide elision control facilities.

Temporal control techniques allow the user to change the mapping between execution time and real time. The most common technique is *speed* where the user can make the program stop and start as well as run faster or slower. Most of the systems at least allow the user to stop and start the visualized program while BALSA, Zeus, Pascal Genie, and UWPI all have an explicit speed control. Reversing the *direction* of time, so that the program runs backwards, is a rare feature as the table shows, but can be extremely useful when trying to understand an algorithm. Even the ability to "rewind to the beginning", such as that provided by TPM, is a useful kind of temporal direction control.

E.3 Scripting Facilities

Does the system provide facilities for managing the recording and playing back of interactions with particular visualizations?

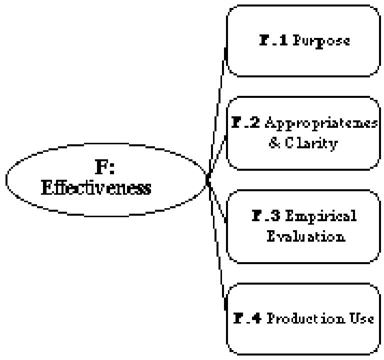
This facility is important when demonstrations are required and it is particularly useful in classroom situations where a demonstration can be run like a videotape or students can go through it at their own pace. Even though many of the systems are designed for novice and expert demonstrations, only BALSA and Zeus have serious scripting support.

F: Effectiveness

How well does the system communicate information to the user?

This is a highly subjective measure and may be made up of many factors. As shown in Figure 26, we see four categories characterizing the effectiveness of a given SV system. Table 6 shows the relative rankings for each of the example systems.

Figure 26



F.1 Purpose

For what purpose is the system suited?

Once one knows what a system is intended to do, one can evaluate how effectively it achieves the intended goal. Many of the example systems are suited to novice classroom demonstration while a few are intended for expert algorithm demonstration or debugging. Few are actually suited to expert software engineering except those tied tightly to the source code.

F.2 Appropriateness and Clarity

If automatic (default) visualizations are provided, how well do they communicate information about the software?

This might also be expressed as: "How rapidly do the visual metaphors inspire understanding?" Among the example systems, we found that those tightly connected to source code, like SEE and ObjectCenter, received the lowest rating, while the others were difficult to distinguish because the quality of their output is highly dependent on individual examples.

F.3 Empirical Evaluation

To what degree has the system been subjected to a good experimental evaluation?

Most SV system developers perform little or no empirical evaluation even though a sound scientific evaluation could prove the effectiveness of their system. One of the reasons that few studies are performed is the poor state of the art in software psychology, where there are no reliable methods for comparing programming environments. In this category we give our opinion on the evaluation reported in the literature (if any) for each system and the relative persuasiveness of the experiment based on the current state of the art. Overall, the number of systems that have been evaluated empirically is low. The systems with the most documented empirical evaluation were TANGO and Pascal Genie; although BALSA, SEE, TPM, and LogoMedia have been subjected to some empirical study, much of it has been informal. A proper empirical study, in addition to convincing people of the efficacy of a system, can also serve to guide system improvements and help define new areas for SV research.

F.4 Production Use

Has the system been in production use for a significant period of time?

This category helps to indicate if people are actually using the system and to what extent. Production use includes publication, sale, and distribution, or consistent use by students in a course. Several of the systems are publicly available over the internet by anonymous ftp (indicated in the references) while others must be purchased from software suppliers. SOS, BALSA, TANGO, Pascal Genie, and TPM have all been released publicly and used in schools and universities to varying extents.

7. Research Agenda

Although the twelve systems illustrate numerous positions along dimensions of the software visualization design space, there are still many outstanding research problems that need to be solved if SV is to aid significantly the practice of professional programming and software engineering as well as the learning of computer science by novice programmers (Eisenstadt, Price, & Domingue, 1993).

The largest impediment to the use of SV by professional programmers is the issue of *scope*. Most SV systems are still dealing primarily with toy programs; many issues of scalability remain to be solved. While small-scale prototypes are useful for exploring new research ideas, there are also valid research issues involved in scaling these ideas up to larger implementations. The challenge now is to implement and test SV techniques on production scale systems. Furthermore, SV has tremendous potential to aid in the understanding of concurrent programs and we expect more fruitful developments in this area as multiprocessors become more common.

The *content* of an SV display can vary widely over the *program* to *algorithm* spectrum. When designing an SV system, it is important to note the system's intended goals and select the content accordingly. For example, systems designed for classroom teaching might intentionally show algorithm-level displays and avoid program-level displays in order to keep the students' minds off implementation details. Systems designed for expert or even novice interactive use might require the ability to move smoothly between algorithm-level and program-level displays depending on their needs. More work is needed to determine how SV systems can perform these kinds of transitions. Control and data flow are both important to software engineers, yet very little work has been done in showing effective transitions between these, especially in a run-time model. A dynamic-to-dynamic temporal control mapping seems to be the overwhelming choice among designers, yet dynamic-to-static mappings have the potential to convey information in a much more concise manner.

Work to date has only scratched the surface in terms of the *form* of software visualizations. We have the beginnings of a diverse graphical and auditory vocabulary for communication, but much of our knowledge in this field is informal. New systems have only just started to explore the use of colour, sound, and multi-dimensional output as a communication medium. We expect a great deal more work to be done in this area, but it should be done in conjunction with *empirical evaluations* and proper psychological studies to determine which techniques are effective. The automatic layout of information is crucial if we are to expand the *scope* of systems as mentioned earlier. Research into the automatic choice of data displays has only just begun (Mackinlay, 1986) and much more needs to be done if there is to be a chance of addressing *scope* concerns. The ability to change levels of *granularity* is often overlooked in SV systems yet it has the potential to help overcome the boundaries of scale by providing a range of coarse- and fine-grained views. Providing *multiple views* of different elements in a visualization is well established and we expect interfaces for controlling multiple views to improve. The ability to compare programs with *synchronized* views provides a powerful teaching tool as well as a performance debugging resource for experts, but to date synchronized views have only been used on toy programs.

The *methods* for specifying software visualizations are quite crude and are a likely reason for the dearth of professional SV systems. If programmers are to use SV systems as program understanding and debugging aids, a great deal more automation must be provided. Otherwise, the effort required to get a visualization will exceed the perceived benefit. Simple automatic displays such as those provided by ObjectCenter or other conventional tools are not enough: some of the power of automatic program understanding and data layout must be employed if the cognitive load on the programmer is to be significantly reduced. Requiring the programmer to understand the code in order to produce a visualization is not appropriate in such cases.

Software visualizations can be very large and complex, both spatially and temporally. *Interaction* with such visualizations requires facilities for advanced *navigation* through large programs and data spaces. We expect to see visualization prototypes which use virtual reality technology to advantage for navigating large, complex spaces. Scripting facilities have not been used extensively in SV systems but their worth has been demonstrated in many kinds of software interfaces, so we expect that production SV systems would include at least rudimentary scripting facilities.

Over one hundred software visualization prototypes have been built in the last twenty years, yet very few of these were systematically evaluated to ascertain their *effectiveness*. The number that have seen any kind of *production use*, particularly in the domain of tools for professional programmers, is particularly small. The most disturbing observation is the lack of proper empirical evaluation of SV systems, for if the systems are not evaluated, what is the point of building them?

If we can make progress with these issues, there are obvious benefits for the fields of software engineering and computer science instruction. Yet the potential goes beyond this to the entire domain of interactive systems, to the users as well as the programmers of interactive systems. Increasingly, the learning and use of complex systems is being facilitated by augmenting conventional textual and still graphic presentations with animation (Baecker & Small, 1990; Baecker, Small, & Mander, 1991), video, and speech and non-speech audio (Mountford & Gaver, 1990). Software visualization can therefore be applied to the development of self-revealing technology that can aid in demystifying and explaining system behaviour to users across the novice to expert continuum.

Acknowledgments

The initial ideas for this paper were presented in a conference paper (Price, Small, & Baecker, 1992), but many changes have since been made and we are grateful to a number of people for their advice and comments on earlier drafts. Marc Eisenstadt, John Domingue, John Stasko, and Chris DiGiano read complete drafts of the paper and helped contribute to the ideas. The three anonymous reviewers provided excellent advice on structure and helped catch some of the errors. The authors and programmers (David Sherman, Marc Brown, John Stasko, Jon Bentley, Brian Kernighan, Brad Myers, Robert Henry, Alan J Rosenthal, Marc Eisenstadt, Gruia-Catalin Roman, Ken Cox, Russell Owen, and Chris DiGiano) of the first eleven SV systems that we presented helped a great deal with corrections to system descriptions and their ratings within the taxonomy, but any remaining errors are ours. We gratefully acknowledge support from the UK SERC/ESRC/MRC Joint Council Initiative on Cognitive Science and Human Computer Interaction (Project 90/CS66), the Natural Sciences and Engineering Research Council of Canada, Apple Computer, Inc., the Information Technology Research Centre of

Ontario, and the Institute for Robotics and Intelligent Systems of Canada.

References

Baecker, R. M. (1968). Experiments in On-Line Graphical Debugging: The Interrogation of Complex Data Structures (Summary Only). In *Proceedings of First Hawaii International Conference on the System Sciences,* (pp. 128-129).

Baecker, R. M. (1975). Two systems which produce animated representations of the execution of computer programs. *ACM SIGCSE Bulletin*, **7**: 158-167.

Baecker, R. M. (1981). *Sorting Out Sorting*. narrated colour videotape, 30 minutes, presented at ACM SIGGRAPH ‘81 and excerpted in ACM SIGGRAPH Video Review #7, 1983. Los Altos, CA: Morgan Kaufmann.

Baecker, R. M. & Buchanan, J. W. (1990). A Programmer’s Interface: A Visually Enhanced and Animated Programming Environment. In *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences.*, (pp. 531-540). New York: IEEE Computer Society Press.

Baecker, R. M. & Marcus, A. (1986). Design Principles for the Enhanced Presentation of Computer Program Source Text. In *Proceedings of Human Factors in Computing Systems (CHI ‘88),* (pp. 51-58). New York: ACM Press.

Baecker, R. M. & Marcus, A. (1990). *Human Factors and Typography for More Readable Programs*. Reading, MA: Addison-Wesley.

Baecker, R. M. & Small, I. S. (1990). Animation at the Interface. In B. Laurel (Eds.), *The Art of Human-Computer Interface Design* (pp. 251-267). Reading, MA: Addison-Wesley.

Baecker, R. M., Small, I. S., & Mander, R. (1991). Bringing Icons to Life. In S. P. Roberston, G. M. Olson, & J. S. Olson (Ed.), *Proceedings of Human Factors in Computing Systems (CHI ‘91),* (pp. 1-6). New York: ACM Press.

Baeza-Yates, R., Jara, L., & Quezada, G. (1992). VCC: Automatic Animation of C Programs. In *Proceedings of COMPUGRAPHICS’92*, (pp. 389-397).

Bentley, J. L. & Kernighan, B. W. (1991a). A System for Algorithm Animation. *Computing Systems,* **4**(1): 5-30.

Bentley, J. L. & Kernighan, B. W. (1991b). *A System for Algorithm Animation (Tutorial and User Manual)* (Computing Science Technical Report No. 132). AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

Bentley, J. L. & Kernighan, B. W. (1992). ANIM. Murray Hill, NJ: AT&T Bell Laboratories. a collection of ANSI C programs used to visualize programs, runs on any UNIX computer. available by anonymous ftp from research.att.com in /netlib/research.

Bertin, J. (1983). *Semiology of Graphics: diagrams, networks, maps* (W. J. Berg, Trans.). Madison, WI:

University of Wisconsin Press.

Bessant, C. (1991). Micro Interpreter for Knowledge Engineering (MIKE). (ver. 2.50) Milton Keynes, UK: Human Cognition Research Laboratory, The Open University. knowledge engineering toolkit written in Prolog, runs on MS-DOS. available by anonymous ftp from hcrl.open.ac.uk in /pub/software.

Brayshaw, M. (1990). *Visual Models of PARLOG Execution* (Technical Report No. 64). Human Cognition Research Laboratory, The Open University, Milton Keynes, England.

Brayshaw, M. (1991). An Architecture for Visualising the Execution of Parallel Logic Programming. In J. Mylopoulos & R. Reiter (Ed.), *Proceedings of The Twelth International Joint Conference on Artificial Intelligence (IJCAI-91)*, 2 (pp. 870-876). San Mateo, CA: Morgan Kaufmann.

Brayshaw, M. & Eisenstadt, M. (1991). A Practical Graphical Tracer for Prolog. *International Journal of Man-Machine Studies*, **35**(5): 597-631.

Brown, M. H. (1988a). *Algorithm Animation*. New York: MIT Press.

Brown, M. H. (1988b). Exploring Algorithms Using Balsa II. *IEEE Computer*, **21**(5): 14-36.

Brown, M. H. (1991). Zeus: A System for Algorithm Animation and Multi-View Editing. In *Proceedings of IEEE Workshop on Visual Languages*, (pp. 4-9). New York: IEEE Computer Society Press.

Brown, M. H. (1992). *Zeus: A System for Algorithm Animation and Multi-view Editing* (Research Report No. 75). DEC Systems Research Center, Palo Alto, CA.

Brown, M. H. & Hershberger, J. (1991). Color and Sound in Algorithm Animation. *Computer*, **25**(12): 52-63.

Brown, M. H., Myrowitz, N., & van Dam, A. (1983). Personal Computer Networks and Graphical Animation: Rationale and Practice for Education. *ACM SIGCSE Bulletin*, **15**(1): 296-307.

Brown, M. H. & Najork, M. A. (1993). *Algorithm Animations Using 3D Interactive Graphics* (Technical Report). DEC Systems Research Centre, Palo Alto, CA.

Brown, M. H. & Sedgewick, R. (1984a). Progress Report: Brown University Instructional Computing Laboratory. *ACM SIGCSE Bulletin*, **16**(1): 91-101.

Brown, M. H. & Sedgewick, R. (1984b). A System for Algorithm Animation. In *Proceedings of ACM SIGGRAPH '84*, (pp. 177-186). New York: ACM.

Brown, M. H. & Sedgewick, R. (1985). Techniques for Algorithm Animation. *IEEE Software*, **2**(1): 28-39.

BSD Unix Distribution. (1988). vgrind. Berkley, CA: University of California at Berkeley. C program, runs on BSD Unix or derivatives. available with most UNIX system software.

Buchanan, J. W. (1988). *LOGOmotion: A Visually Enhanced Programming Environment*. M.Sc. Thesis, Dept. of Computer Science, University of Toronto, M5S 1A4.

CenterLine Software (1991). *ObjectCenter Reference*. Cambridge, MA: CenterLine Software, Inc.

Chandhok, R., Garlan, D., Meter, G., Miller, P., & Pane, J. (1991). Pascal Genie. (ver. version 1.0) San Diego, CA: Chariot Software Group. Pascal programming environment with integrated structure-driven editor, runs on Macintosh Computers. available from Chariot Software Group, San Diego, CA.

Chang, S.-K. (Ed.). (1990). *Principles of Visual Programming Systems*. EngleWood Cliffs, NJ: Prentice-Hall.

Conrow, K. & Smith, R. G. (1970). NEATER2: A PL/I Source Statement Reformatter. *Communications of the ACM*, **13**(11): 669-675.

Cypher, A. (Ed.). (1993). *Watch What I Do: Programming by Demonstration*. Cambridge, MA: MIT Press.

DiGiano, C. J. (1992). *Visualizing Program Behavior Using Non-speech Audio*. M.Sc. Thesis, Dept. of Computer Science, The University of Toronto, M5S 1A4, Canada.

DiGiano, C. J. & Baecker, R. M. (1992). Program Auralization: Sound Enhancements to the Programming Environment. In *Proceedings of Graphics Interface'92*, (pp. 44-53). Palo Alto, CA: Morgan Kaufmann.

DiGiano, C. J., Owen, R., & Rosenthal, A. J. (1992). LogoMedia. Toronto, Canada: Department of Computer Science. Logo Programming Environment supporting audio visualization, runs on Macintosh II compatibles. available by anonymous ftp from hcrl.open.ac.uk in /pub/software.

Eisenstadt, M. (Ed.). (1988). *Intensive Prolog, Associate Student Office (Course PD622)*. Milton Keynes (UK): Open University Press.

Eisenstadt, M. & Brayshaw, M. (1986). *The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming* (Technical Report No. 21). Human Cognition Research Laboratory, The Open University, Milton Keynes, MK7 6AA, UK.

Eisenstadt, M. & Brayshaw, M. (1988). The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, **5**(4): 1-66.

Eisenstadt, M. & Brayshaw, M. (1990). A Knowledge Engineering Toolkit, part I. *BYTE: The Small Systems Journal*, **10**(10): 268-282.

Eisenstadt, M., Brayshaw, M., & Payne, J. (1991). *The Transparent Prolog Machine: visualizing logic programs*. Dordrect, The Netherlands: Kluwer Academic Publishers.

Eisenstadt, M., Domingue, J., Rajan, T., & Motta, E. (1990). Visual Knowledge Engineering. *IEEE Transactions on Software Engineering*, **16**(10): 1164-1177.

Eisenstadt, M., Price, B. A., & Domingue, J. (1993). Software visualization as a pedagogical tool. *Instructional Science*, **21**: 335-365.

Flinn, S. & Cowan, W. (1990). Visualizing the Execution of Multi-processor Real-Time Programs. In *Proceedings of Graphics Interface*, (pp. 293-300). Halifax, Nova Scotia:

Francioni, J. M., Albright, L., & Jackson, J. A. (1992). Debugging parallel programs using sound. *SIGPLAN Notices*, **26**(12): 68-75.

Gaver, W., O'Shea, T., & Smith, R. (1991). Effective Sounds in Complex Systems: The ARKola SImulation. In S. P. Roberston, G. M. Olson, & J. S. Olson (Ed.), *Proceedings of Human Factors in Computing Systems (CHI '91)*, (pp. 85-90). New York: ACM Press.

Glinert, E. P. (Ed.). (1990a). *Visual programming environments : applications and issues*. New York: IEEE Computer Society Press.

Glinert, E. P. (Ed.). (1990b). *Visual programming environments : paradigms and systems*. New York: IEEE Computer Society Press.

Goldenson, D. R. (1989). The Impact of Structure Editing on Introductory Computer Science Education: The Results So Far. *ACM SIGCSE Bulletin*, **21**(3): 26-29.

Goldstein, H. H. & von Neumann, J. (1947). Planning and Coding Problems of an Electronic Computing Instrument. In A. H. Taub (Eds.), *von Neumann, J., Collected Works* (pp. 80-151). New York: McMillan.

Haibt, L. M. (1959). A Program to Draw Multi-Level Flow Charts. In *Proceedings of The Western Joint Computer Conference*, 15 (pp. 131-137). San Francisco, CA:

Heltulla, E., Hyskykari, A., & Räihä, K.-J. (1989). Graphical Specification of Algorithm Animations with ALLADIN. In *Proceedings of The Twenty-Second Annual Hawaii International Conference on System Science*, (pp. 892-901). New York: IEEE Computer Society.

Henry, R. R., Whaley, K. M., & Forstall, B. (1990a). The University of Washington Illustrating Compiler. In *Proceedings of The ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, (pp. 223-233). New York: ACM.

Henry, R. R., Whaley, K. M., & Forstall, B. (1990b). The University of Washington Program Illustrator (UWPI). Seattle, WA: Computer Science and Engineering, University of Washington. Common Lisp/CLX source code for an automatic Pascal algorithm animation system, runs on Unix Workstations. available by anonymous ftp from june.cs.washington.edu as pub/uwpi.tar.Z.

Hueras, J. & Ledgard, H. (1977). An Automatic Formatting Program for Pascal. *ACM SIGPLAN Notices*, **12**(7): 82-84.

Knowlton, K. C. (1966a). *L[6]: Bell Telephone Laboratories Low-Level Linked List Language*. 16 mm black and white sound film, 16 minutes. Murray Hill, NJ: Technical Information Libraries, Bell Laboratories, Inc.

Knowlton, K. C. (1966b). *L[6]: Part II. An Example of L[6] Programming*. 16 mm black and white sound film, 30 minutes. Murray Hill, NJ: Technical Information Libraries, Bell Laboratories, Inc.

Knowlton, K. C. (1966c). A Programmer's Description of L[6]. *Communications of the ACM*, **9**(8): 616-625.

Knuth, D. E. (1963). Computer-Drawn Flowcharts. *Communications of the ACM*, **6**(9): 555-563.

Knuth, D. E. (1984). Literate Programming. *The Computer Journal*, **27**(2): 97-111.

Kraemer, E. & Stasko, J. T. (1993). The Visualization of Parallel Systems: an overview. *Journal of Parallel and Distributed Computing*(May): to appear.

Kwakkel, F. (1991). TPM for Macintosh. (ver. 1.1) Milton Keynes, UK: Human Cognition Research Lab, The Open University. Prolog Interpreter and graphical tracer, runs on Macintosh II. available by anonymous ftp to hcrl.open.ac.uk.

Ledgard, H. F. (1975). *Programming Proverbs*. Rochell Park, NJ: Hayden.

Lieberman, H. (1984). Steps Toward Better Debugging Tools for Lisp. In *Proceedings of Third Lisp Conference*,

Lieberman, H. (1989). A Three-Dimensional Representation For Program Execution. In *Proceedings of The 1989 IEEE Workshop on Visual Languages*, (pp. 111-116). New York: IEEE Computer Society Press.

London, R. L. & Duisberg, R. A. (1985). Animating Programs using Smalltalk. *Computer*, **18**(8): 61-71.

Mackinlay, J. (1986). Automating the Design of Graphical Presentations of Relational Information. *ACM Transactions on Graphics,* **5**(2): 110-141.

Martin, J. & McClure, C. (1985). *Diagramming Techniques for Analysts and Programmers*. Englewood Cliffs, NJ: Prentice-Hall.

Miara, R. J., Musselman, J. A., Navarro, J. A., & Shneiderman, B. (1983). Program Indentation and Comprehensibility. *Communications of the ACM*, **26**(11): 861-867.

Mountford, S. J. & Gaver, W. W. (1990). Talking and Listening to Computers. In B. Laurel (Eds.), *The Art of Human-Computer Interface Design* (pp. 319-334). Reading, MA: Addison-Wesley.

Myers, B. A. (1980). *Displaying Data Structures for Interactive Debugging* (Technical Report No. CSL-80-7). Xerox PARC.

Myers, B. A. (1983). Incense: A System for Displaying Data Structures. *Computer Graphics,* **17**(3): 115-125.

Myers, B. A. (1986). Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In M. Mantei & P. Orbeton (Ed.), *Proceedings of Human Factors in Computing Systems*

(CHI '86), (pp. 59-66). New York: ACM Press.

Myers, B. A. (1988). *The State of the Art in Visual Programming and Program Visulization* (Technical Report No. CMU-CS-88-114). Computer Science Dept., Carnegie-Mellon University, Pittsburg, PA.

Myers, B. A. (1990). Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, **1**(1): 97-123.

Myers, B. A., Chandhok, R., & Sareen, A. (1988). Automatic Data Visualization for Novice Pascal Programmers. In *Proceedings of The IEEE Workshop on Visual Languages,* (pp. 192-198). New York: IEEE Computer Society Press.

Nassi, I. & Shneiderman, B. (1973). Flowchart Technique for Structured Programming. *ACM SIGPLAN Notices,* **8**(8): 12-26.

Norcio, A. F. (1982). Indentation, Documentation, and Programmer Comprehension. In *Proceedings of Human Factors in Computing Systems (CHI '82),* (pp. 118-120). New York: ACM Press.

Payne, J. (1991). TPM for Sun. Cambridge, UK: Expert Systems Ltd. Interpreter and Graphical Tracer for Prolog, runs on Sun Microsystems Machines. available from Expert Systems Ltd., Cambridge, UK.

Price, B. A. (1990). *A Framework for the Automatic Animation of Concurrent Programs*. M.Sc. Thesis, Dept. of Computer Science, University of Toronto, Canada M5S 1A1.

Price, B. A. & Baecker, R. M. (1991). The Automatic Animation of Concurrent Programs. In *Proceedings of The First International Workshop on Computer-Human Interfaces,* (pp. 128-137). Moscow, USSR: ICSTI.

Price, B. A., Small, I. S., & Baecker, R. M. (1992). A Taxonomy of Software Visualization. In *Proceedings of The 25th Hawaii International Conference on System Sciences,* II (pp. 597-606). New York: IEEE Computer Society Press.

Reiss, S. P. (1990). Interacting with the FIELD Environment. *Software Practice and Experience,* **20**(S-1): 89.

Roman, G.-C. & Cox, K. C. (1989). A Declarative Approach to Visualizing Concurrent Computations. *Computer*, **22**(10): 25-36.

Roman, G.-C., Cox, K. C., & Boemker, T. J. (1990). *Diffusing Computations*. 7 minute silent, colour S-VHS videotape. St. Louis, MO 63130-4899: Dept. of Computer Science, Washington University.

Roman, G.-C., Cox, K. C., Plun, J. Y., & Wilcox, C. D. (1992a). *From Proofs to Pictures*. 14 minute colour S-VHS videotape with sound. St. Louis, MO 63130-4899: Dept. of Computer Science, Washington University.

Roman, G.-C., Cox, K. C., Wilcox, C. D., & Plun, J. Y. (1992b). Pavane: a System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing*, **3**(2): 161-193.

Roman, G.-C. & Cunningham, H. C. (1990). Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency. *IEEE Transactions on Software Engineering*, **16**: 1361-1373.

Roy, P. & St. Denis, R. (1976). Linear Flowchart Generator for a Structured Language. *ACM SIGPLAN Notices*, **11**(11): 58-64.

Scanlan, D. A. (1989). Structured Flowcharts Outperform Pseudocode: An Experimental Comparison. *IEEE Software*, **6**(5): 28-36.

Sedgewick, R. (1983). *Algorithms*. Reading, MA: Addison-Wesley.

Shu, N. C. (1988). *Visual Programming*. New York: Van Nostrand Reinhold.

Simpson, J. A. & Weiner, E. S. C. (Ed.). (1989). *The Oxford English Dictionary* (2nd ed.). Oxford: Oxford University Press.

Small, I. S. (1989). *Program Visualization: Static Typographic Visualization in an Interactive Environment*. M.Sc. Thesis, Dept. of Computer Science, University of Toronto.

Stasko, J. T. (1989). *TANGO: A Framework and System for Algorithm Animation* (Technical Report No. CS-89-30). Brown University, Providence, RI 02912.

Stasko, J. T. (1990). Tango: A Framework and System for Algorithm Animation. *IEEE Computer*, **23**(9): 27-39.

Stasko, J. T. (1991). Using Direct Manipulation to Build Algorithm Animations by Demonstration. In S. P. Robertson, G. M. Olson, & J. S. Olson (Ed.), *Proceedings of Human Factors in Computing Systems (CHI '91)*, (pp. 307-314). New York: ACM Press.

Stasko, J. T. (1992). XTango. (ver. 1.42) Atlanta, GA: College of Computing, Georgia Institute of Technology. C source code with example scripts, runs on UNIX workstations running the X Window System. available by anonymous ftp from par.cc.gatech.edu in /pub.

Stasko, J. T. & Patterson, C. (1992). Understanding and Characterizing Software Visualization Systems. In *Proceedings of IEEE 1992 Workshop on Visual Languages*, (pp. 3-10). New York: IEEE Computer Society Press.

Stasko, J. T. & Whrli, J. F. (1992). *Three-Dimensional Computation Visualization* (Technical Report No. GIT-GVU-92-20). Graphics, Visualization, and Usability Center, College of Computing, Georgia Institue of Technology, Atlanta, GA 30332-0280.

Weizenbaum, J. (1966). Eliza--A Computer Program for the Study of Natural Language Communication Between Man and Machine. *Communications of the ACM*, **9**(1): 36-45.

Zimmermann, M., Perrenoud, F., & Schiper, A. (1988). Understanding Concurrent Programming through Program Animation. In H. L. Dershen (Ed.), *Proceedings of The Nineteenth ACM SIGCSE Technical Symposium on Computer Science Education*, (pp. 27-35).

B.A.Price@open.ac.uk 14 July 94