A Taxonomy of Software Visualization

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

Dynamic Graphics Project, Computer Systems Research Institute
University of Toronto, CANADA M5S 1A1
Phone: +1 416-978-6983, Fax: +1 416-978-5184
Internet: rmb@dgp.toronto.edu

Abstract

Software visualization is the use of interactive computer graphics, typography, graphic design, animation, and cinematography to enhance the interface between the software engineer or the computer science student and their programs. Although several taxonomies of software visualization have been proposed, they use few dimensions and do not span the space of important distinctions between systems. We attempt to fill this gap in the literature by proposing a novel and systematic taxonomy of six areas making up thirty characteristic features of software visualization technology. The taxonomy is presented and illustrated in terms of its application to seven systems of historic importance and technical interest.

Introduction

Through scientific visualization, researchers across a range of scientific disciplines can take advantage of new hardware and software technology to display and elucidate vast quantities of otherwise incomprehensible data (McCormick, DeFanti and Brown 1987; Rosenblum and Nielson 1991). Since the data is presented in a pictorial form, scientists are able to use the brain's ability to make analogies and links between the visual image and existing ideas—links that are not likely to be made when data appears as columns of numbers or lines of text. A good scientific visualization system allows the researcher to make discoveries not otherwise possible and provides him with a powerful new interface to his data.

Ironically, the field of software engineering has not seen the benefits of the scientific visualization revolution. Software engineers also deal with large amounts of complex data (computer programs), but they still rely on an interface which has remained essentially unchanged for twenty years. Programs are often edited in variably-sized overlapping windows on large screen desktop workstations, but the programs still appear in single-colour, single-font text. While the amount of screen real estate has increased, the fundamental interface is that of the VT-100 style terminal of the 1970's.

Programming language researchers have been trying to design languages that make it easier to express and understand programs, but they have been restricted by the Roman alphabet and the ASCII character set, the elements of which are often difficult to discriminate and take limited advantage of the human visual perception system. The field of computer *software visualization* (SV) has a similar goal, but is not handicapped by these restrictions: it facilitates the human understanding and effective use of computer programs by relying on the crafts of typography, graphic design, animation, cinematography, and interactive computer graphics.

The importance of visual representations in understanding computer programs is by no means a new concept. Goldstein and von Neumann (1947) demonstrated the usefulness of flowcharts, while Haibt (1959) developed a system that could draw them automatically. But flowcharts have serious limitations and few software engineers would argue that flowcharts significantly enhance their understanding of million line systems. Researchers started to address this problem in the 1980's, as workstation technology became more common, encouraging the development of a broader range of SV techniques. This work ranged from motion pictures of sorting algorithms (Baecker, 1981) to fully interactive graphical representations of data structures running "live" on a workstation (Brown and Sedgewick, 1984). Yet this technology remains largely unused by today's software engineer.

A common reason cited for the failure of SV systems as software engineering tools is their lack of scalability. Many systems only work for fixed examples, "toy" programs, or subsets of a language. Several require extensive modification of the source code, thus making them inappropriate for large software engineering projects. Some only display one kind of data or one level of abstraction, which make them appropriate only for certain applications. Few systems allow the user to navigate easily

*Author's current address: Human Cognition Research Laboratory, The Open University, Milton Keynes, UK, MK6 2DQ. Internet: ba_price@vax.acs.open.ac.uk

To appear in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, January 1992.

through the large information space of a software project or provide the detail suppression necessary to grasp the larger concepts.

We believe that this failure of technology transfer is due, in part, to the lack of systematic organization within which researchers have positioned their systems. There are many characteristics that make a software engineering tool "useful" and the optimal values of these characteristics vary depending on the situation. Previous taxonomies (e.g., Myers, 1990; Eisenstadt, Domingue, Rajan, and Motta, 1990) have suggested a handful of criteria for comparing SV systems. While these are sufficient for a general taxonomy, they are not detailed enough to evaluate the effectiveness and breadth of application of a particular system.

In this paper, we present a comprehensive set of thirty characteristics for evaluating a SV system. We argue that these characteristics span the space of important distinctions between systems and allow us to discover where previous systems have succeeded or failed. We apply these characteristics to seven important SV systems to illustrate the use of the taxonomy to categorize different approaches. We conclude by describing a research agenda for software visualization and by identifying important characteristics for a successful software engineering tool.

Terminology

What we are calling *software visualization* has often been called *program visualization* in the literature. We prefer the former term since it encompasses algorithms, programs, and data, and since it suggests the need to deal with systems consisting of multiple programs.

Program visualization, *visual programming*, and *programming by example* are sometimes confused with each other. Visual programming is the *specification* of a computer program using graphics (Shu, 1988) while program visualization is the use of graphics to *enhance the understanding* of a program that has already been written. Programming by example involves specifying a program by giving examples of the input and output data and having the computer infer the program (Myers, 1990). There is certainly some overlap between these areas, since a program specified with graphical examples is a kind of visual programming, and a graphical program specification will likely improve one's understanding of the program, at least to some degree. This paper will focus on SV tools that are not also visual programming systems.

Seven Software Visualization Systems

To illustrate our taxonomy, we shall apply each of the categories to seven different SV systems. We chose these systems because of their historic importance and because they illustrate a diversity of approaches. These systems do not include all of 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.

The first major work of the 1980's was the motion picture *Sorting Out Sorting* (Baecker, 1981), produced at the University of Toronto. This 30 minute colour, narrated educational film/videotape uses animated computer graphics to explain how nine different sorting algorithms manipulate their data. The movie begins by showing how the algorithms manipulate a handful of data items and ends with a race of all nine algorithms simultaneously sorting sets of two thousand elements each. It is still used today as a teaching tool.

BALSA (Brown and Sedgewick, 1984), the first widely used interactive SV system, was used extensively in the Brown University Workstation Project as an educational tool for teaching introductory computer science courses. This interactive, workstation-based system allows users to watch a graphical, high level representation of data structures in a running Pascal program. The current statement being executed is highlighted in the source code and each data structure graphic reflects the current contents of the data structure. The user can stop and start the program at any time, as well as control the program speed and run the program backwards. The animation is actually achieved by calls to animation routines which the user has inserted in the source code at locations of "interesting events." The user can also write his own animation routines in Pascal. The most recent version, BALSA II (Brown, 1988), supports colour and rudimentary sound.

The SEE Program Visualizer (Baecker and Marcus, 1990) is a UNIX-based system for typesetting programs written in C. This work combines human factors research with modern typography and laser printing technology to format C programs automatically and produce a kind of "program book" with cross-references and indices that facilitate navigation through the source code.

Bell Laboratory's Movie and Stills (Bentley and Kernighan, 1987) can generate visualizations of programs written in any language under the UNIX operating system. The user must insert print statements into the source code so that Movie language statements are output at appropriate points in the program. To display the animation, the user must first run the program and collect the output in a file. The Movie language output is a program in a "little language" which is then run through the Movie interpreter. This results in a display of the animation of the program.

The Transparent Prolog Machine (TPM) was developed at the Open University (Eisenstadt and Brayshaw, 1988). It is the only one of the seven systems that visual-

izes a logic language. TPM is a Prolog interpreter (running on a workstation) that has been instrumented to display *automatically* a running trace of the program with either coarse-grained or fine-grained views. The coarse-grained view shows the final outcome of attempted goals using a schematized AND/OR tree with each node summarizing the outcome of a call to a particular procedure. The fine-grained view shows a detailed execution history of individual clauses using augmented AND/OR tree (AORTA) diagrams. TPM is designed to scale up from toy examples to much larger programs.

The University of Toronto LogoMotion system (Baecker and Buchanan 1990) produces animations of the execution of programs in the Logo language. This is done almost automatically, except that a user must indicate what aspects (procedures or data) of a Logo program he would like to see. If the automatically produced default visualizations are not suitable, he can augment the portrayal through writing additional code in the host Logo language, but the program being animated need not be touched.

The University of Washington Program Illustrator (UWPI) is a specially instrumented Pascal compiler that has been trained to recognize certain patterns of code and data structure declarations (Henry, Whaley, and Forstall, 1990). This system will automatically compile programs written in a subset of Pascal and use the X Window System to display a high level graphical view of certain data structures as the program runs. UWPI currently recognizes generalized data structures for graph, linked list, and sorting algorithms.

The Taxonomy Explained

The taxonomy we propose for characterizing program visualization systems is divided into six major sections: Scope, Content, Form, Method, Interaction, and Effectiveness. Each of these major sections is then further subdivided; the categories which result are called *characteristics* because each of them characterizes a program visualization system in a particular way. Each characteristic is described briefly within the framework of the six major sections.

A. Scope

What are its general characteristics?

1. System/Example

Is the work discussed a visualization system or an example of visualization? Visualization systems can generate visualizations of arbitrary programs within some particular class. Examples of visualization are either hand-coded demonstrations or flexible visualizations of a particular algorithm, system, or set of programs. While examples have a fixed application, they may be quite flexible in the type of visualization they can produce for the entity that they visualize.

2 Class of Program

What class of program is the system designed to visualize? The class can be characterized by such attributes as source language, operating system or environment, or application. Systems designed for one class are domain specific and may not work well for programs in another class.

3. Scalability

What restrictions are there on working with large programs or datasets? This characterization merges design issues (were the visualizations designed to scale well?), fundamental limitations (does the system have any preset or inherent limits on the size of the program being visualized?) and practical experience (has there been any large program experience reported?) into a single category. This characterization is very important because many visual tools only support toy systems—a primary criticism of the genre.

4. Multiple Programs

Can the system generate 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 will allow almost any window-based visualization system to be run in parallel; this approach is not considered satisfactory because there is no centralized control or synchronization between both running visualizations.

5. Concurrency

Does the system support the visualization of concurrent programs? This is an important issue because concurrent applications present special visualization needs which require specialized support within the visualization system.

6. Benign/Disruptive

If the system can be used to visualize concurrent applications, does its use disrupt the execution sequence of the program, or does it have no effect on the concurrent execution sequence? 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.

B. Content

What gets visualized?

7. Program/Algorithm Visualization

Is the system designed to produce algorithm or program visualization? The differentiation 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 labelled 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.

8. Code Visualization

If the system performs program visualization, can it visualize the program code? 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.

9. Data Visualization

If the system performs program visualization, can it visualize the program data? Systems that can visualize data will differ in the extent to which they can gracefully depict complex data structures.

10. Compile/Run-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 are limited to visualizing the program code and its data structures. The system cannot produce any visualization of the actual data values, since it does 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.

11. 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.

C. Form

What elements are used in the visualization?

12. 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 NeWS or Display PostScript), we only list the primary target medium. Common choices include paper, film or videotape, terminal and workstation.

13. Graphical Elements

What graphical elements are used in the visualization produced by the system? This provides some idea of the complexity of the visual primitives which make up the system's graphical vocabulary. Vocabularies can range from simple ones which feature combinations of dots and lines to more complex sets of 2D graphical elements, which could include rectangles, lines, circles, points, etc., all of which may have arbitrary attributes for thickness, fill pattern, etc.

14. Colour

Does the system make use of colour in its visualizations? Colour can be used to convey a great deal of information while imposing a low cognitive load and it has been greatly underutilized 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.

15. Animation

If the system gathers run-time data, is the resulting visualization animated or static? The most obvious and frequent use of animation in program visualization systems is to capture and convey the temporal aspects of the program in execution. Does the system make use of animation in any other novel ways?

16. Multiple Views

Does the system provide multiple synchronized views of different parts of the software being visualized? These

might include 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.

17. Other Modalities

While graphical elements, colour and animation can categorize the purely visual aspects of the displays provided by the system, are there any other modalities used by the system? These modalities would appeal to senses other than sight: hearing, touch, smell and taste. Both speech and non-speech audio appeal to the auditory senses, and have been used in some visualization systems. Of the remaining senses, researchers (Brooks, 1989) are currently working on computer-generated textures which can be touched, while the entertainment industry has dabbled in smell through such vehicles as Sensorama (Fisher, 1990).

D. Method

How is the visualization specified?

18. 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); can be produced by *modifying the source code* and adding visualization code to the original program; can be described by writing *additional visualization code* which operates in harmony with an existing program (but does not actually require modifications to the original source code); can be based on the feedback obtained from *probes* attached to various points or structures in the program; or can be produced completely *automatically*, in which case the system creates a visualization based purely on its analysis of the program, thus requiring no user intervention.

19. Batch/Live

If the visualization data is gathered at run-time, is the visualization produced as a batch job from data recorded during a previous run, or is it produced live as the program executes? If the visualization is live, the user's input can result in an immediate change in the visualization. If the visualization is based on "batch" data extracted from some sort of trace file, the user cannot affect the course of the execution of the program.

20. Fixed/Customizable

Is the visualization which is generated completely fixed, or can the user customize it by some means? Note that being able to visualize different data sets, which is implied by a live categorization above, does not qualify as customizing the visualization. In order to customize the visualization, the layout or presentation of the visualization must be changeable by explicit user instruction.

21. Code Familiarity

If the visualization system is not completely automatic, how much knowledge of the program code is required for a visualization to be produced? Clearly, completely automatic visualization requires no prior knowledge on the part of the user, 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 to which users can attach visualization code may require knowledge of the program if the user wishes to make the best use of the potential "probes" available.

22. Invasive

Does the program source code have to be modified in order to obtain a visualization? Systems which require or encourage the user to write visualization code often rely on the user to place appropriate statements calling the visualization modules from within the program to be visualized. Other systems depend on interpreters to allow them to catch and flag events of interest within the executing program. Some systems instrument a compiler to automatically modify the source code as it is being compiled; these systems are not considered to be invasive, as they do not require the user to go in to modify the source program, but rely on the compiler to do it automatically instead.

23. Customization Language

If the visualization is customizable, how can the visualization be specified? Systems which support *interactive manipulation* of the visualization have their visualizations specified interactively through direct manipulation. Systems which require the user to program explicit visualization code rely on *procedural visualizations*. Systems which allow the user to describe the desired visualization using high-level tools support *declarative specification*. Visualization systems can easily support more than one of these approaches for different aspects of the complete visualization specification.

24. Same Language

If the visualization is specified procedurally, is the specification written in the same programming language as the program to be visualized? A system which seeks general application would not feature this property, while for a system which is designed to work very well within a single programming environment, not requiring the user to learn a new language can be a significant win.

E. Interaction

How do you interact with and control the visualization?

25. Navigation

How well does the system support navigation through a visualization of a large program or dataset? How carefully has the visualization been designed for such situations? Does the visualization support large systems well, or does its interface suffer as the scale of the program being visualized increases? Eisenstadt et al. (1990) suggest that navigability may be achieved by changes of resolution, scale, compression, selectivity, and abstraction.

26. Elision

Does the system support techniques for eliding information or suppressing detail from the display? Such 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. Elision is of primary use with large problems, for which the entire dataset cannot be simultaneously displayed.

27. Temporal Control Mapping

If the visualization is based on data gathered at run-time, what is the mapping between "program time" and "visualization time"? If the visualization is based on information gathered at a single point in time during the program's execution, and generates a static visualization, its mapping is "static to static"; the system generates a *snapshot*. If the visualization generated is instead animated, the mapping is "static to dynamic"; we do not know of any examples of such systems. If the visualization gathers information of 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*. If the visualization system maps "dynamic to dynamic", it uses information gathered over a period of time during the program's execution to generate an *animation*.

F. Effectiveness

How good is the visualization?

28. Appropriateness and Clarity

How well does the visualization communicate information about the software? How rapidly do the visual metaphors inspire understanding? Example visualizations or systems that provide *defaults* for automatically generated presentations can be judged in these terms. Appropriateness and clarity is a subjective measure and may need to be validated by experimentation or by the "marketplace." We have declined to make these subjective judgements here without having the space for a full explanation; see Small, Baecker, and Price (in preparation) for a more complete treatment.

29. Experimental Evaluation

Has the system been subjected to a good experimental evaluation? See, for example, Baecker and Buxton (1987), Case Study B, for an introduction to the field of software psychology.

30. Production Use

Has the system been in production use for a significant period of time? This includes consistent use by students in a course or publication, sale, and distribution.

Analyzing the Seven Systems Using the Taxonomy

If the taxonomy provides a meaningful way of describing software visualization technology, then it should facilitate a clear and concise statement of the essential features, similarities, and differences between specific systems, and it should encourage the development of insights into the weaknesses of existing technologies and the needs for new technologies. We shall first apply it to the seven systems, then use it in the next section to suggest a research agenda for software visualization.

The following six tables present the application of the taxonomy to the seven systems. If a table entry appears to be missing, refer back to the description of the taxonomy characteristic which states the conditions under which that characteristic is meaningful. Space constraints in this paper force some oversimplification; the reader is referred to Small, Baecker, and Price (in preparation) for a more complete exposition.

We can see from the tables that the taxonomy does illuminate and clarify the major roles and distinctions among the seven systems.

Sorting Out Sorting demonstrates the use of algorithm animation to explain nine specific algorithms. Through multiple program animation, and through careful attention to visual techniques for data scalability, it effectively contrasts these algorithms in terms of their performance. A rudimentary use of colour and sound add significantly to the comprehensibility of the presentation.

Balsa begins where SOS left off, providing an environment for generating animations of arbitrary Pascal programs. The presentations appear on a workstation, and are live and customizable, through modifications of and additions to the host Pascal program.

A. *Scope*	**1 System/ Example**	**2 Program Class**	**3 Scalability**	**4 Multiple Programs**	**5 Concurrency**	**6 Benign/ Disruptive**
SOS	Example	Specific sorting algorithms	Fixed algorithm set; shows large data sets	Yes	No	
BALSA II	System	Pascal	No built-in support for large data sets	Yes	No	
SEE	System	C; limited preprocessor support	Some support; handles large programs	No	No	
M & S	System	Unix applications	No built-in support	No	Nominal	Disruptive
TPM	System	Prolog	Built-in support; interpreter-based	No	No	
LogoMotion	System	Logo	No built-in support; interpreter-based	No	No	
UWPI	System	Subset of Pascal; limited algorithm set	No large program experience	No	No	

B. *Content*	**7 Program/ Algorithm**	**8 Code**	**9 Data**	**10 Compile-/ Run-Time**	**11 Fidelity and Completeness**
SOS	Algorithm			Run	Partial
BALSA II	Both	Some	Yes	Run	
SEE	Program	Yes	No	Compile	Yes
M & S	Both	No	Yes	Run	
TPM	Program	Yes	Yes	Run	Yes
LogoMotion	Both	No	Yes	Run	
UWPI	Program	No	Yes	Run	Partial

C. *Form*	**12 Medium**	**13 Graphical Elements**	**14 Colour**	**15 Animation**	**16 Multiple Views**	**17 Other Modalities**
SOS	Film	Rectangles, lines and dots	Yes	Yes	No	Sound
BALSA II	Workstation	2D graphic primitives and text	Yes	Yes	Yes	Sound
SEE	Paper	Multi-font typeset text, rules and grey tone	No	No	No	None
M & S	Workstation and paper	2D graphic primitives and text	No	Yes	Yes	None
TPM	Workstation and paper	2.5D graphic primitives and text	Yes	Yes	Yes	None
LogoMotion	Workstation	2D graphic primitives and text	No	Yes	Yes	None
UWPI	Workstation	2D graphic primitives and text	No	Yes	No	None

D. Method	18 Specification Style	19 Batch/ Live	20 Fixed/ Custom.	21 Code Familiarity	22 Invasive	23 Customization Language	24 Same Lang.
SOS	Hand-coded	Batch	Fixed				
BALSA II	Modified source & visualization code	Live	Custom.	Extensive	Yes	Procedural	Yes
SEE	Automatic		Custom.		No	Declarative	
M & S	Modified source & visualization code	Batch	Custom.	Extensive	Yes	Procedural	No
TPM	Automatic	Live	Custom.		No	Interactive	
LogoMotion	Probes & visualization code	Live	Custom.	Minimal	No	Procedural	Yes
UWPI	Automatic	Live	Fixed		No		

E. Interaction	25 Navigation	26 Elision	27 Temporal Control Mapping
SOS		No	Animation
BALSA II	Minimal	No	Animation
SEE	Some	Some	
M & S	No	No	Snapshot, Trace, Animation
TPM	Yes	Yes	Trace, Animation
LogoMotion	No	No	Animation
UWPI	No	No	Animation

F. Effectiveness	28 Appropriate-ness and Clarity	29 Experimental Evaluation	30 Production Use
SOS	subjective	no	yes
BALSA II		no	some
SEE	subjective	some	no
M & S		no	some
TPM	subjective	some	yes
LogoMotion		no	no
UWPI	subjective	no	no

The SEE visualizing compiler illustrates a very different approach, gathering information at compile-time to present typographically-enhanced source code. The form of the visualization may be specified through use of a declarative language.

Like Balsa, Movie and Stills produces program animation, but it allows the animation of programs in any UNIX language. As with Balsa, the source must be augmented with animation calls; unlike Balsa, this results in the production of an intermediate file in a "little language," which is then interpreted in a batch mode to produce either a movie or a series of printed still snapshots.

The Transparent Prolog Machine demonstrates automatic software visualization and its applicability to a different style of language, a logic language. Careful attention to issues of data scalability and navigation allow use of the system with programs of significant size.

LogoMotion illustrates the "probe" method of specifying visualizations. Users indicate what aspects (procedures or data) of a Logo program they would like to see. If the automatically produced default visualizations are not suitable, they can customize the portrayal by writing additional Logo code without touching the host program.

The Washington Program Illustrator is another automatic system. Fixed visualizations tailored to specific problem domains are produced through the incorporation of a knowledge base tailored to the domain.

A Research Agenda

Although the seven systems illustrate many positions along the dimensions of the program visualization design space, a closer examination reveals that there are still many outstanding research issues that need to be solved if software visualization is to aid significantly the practice of software engineering.

The largest failure of work to date is in terms of *scope*. We are still dealing primarily with toy programs; many issues of scalability remain to be solved. There are no known cases of running and comparing several versions of real programs. The work on concurrency is scattered, has lacked a unifying systematic framework, and has yet to be demonstrated on significantly large systems.

Although there has been some progress in the development of methods for the intelligent presentation of complex *content* (Myers, 1983; Eisenstadt and Brayshaw, 1988), we cannot yet display and automatically lay out

appropriate diagrams of complex data and control structures.

Work to date has only scratched the surface in terms of the *form* of software visualizations. No one has yet systematically explored methods of mapping attributes of algorithms and executing programs into such attributes of colour as hue, saturation, and value. The appropriate use of sound is therefore another area of significant research potential (see, for example, Sonnenwald et al. (1990) and Albright et al. (1991)); many can recite anecdotes of the value of listening to one's program, but little systematic work has been done.

The *methods* of specifying software visualizations have thus far been quite crude. Significant portions of a default visualization should be generated automatically. Visualization and customization languages should employ, where appropriate, procedural specification, direct manipulation, and programming by example.

Complex software visualizations are very large, both spatially and temporally. Much work needs to be done on methods of *interaction* which incorporate capabilities for navigation, elision and detail suppression, and control of the mapping from program time to "visualization time."

Hundreds of program and algorithm visualization prototypes have been built in the last twenty years. Very few of these were systematically evaluated to ascertain their *effectiveness* or resulted in systems that saw regular use or became products. There has also been very little principled analysis of various methods of visual representation and presentation.

If we can make progress with these issues, the potential reward is great. There is an obvious payoff for the field of software engineering. (See the Conventional Wisdom Watch below, with apologies to *Newsweek* magazine.) 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 and Small, 1990; Baecker, Small, and Mander, 1991), video, and speech and non-speech audio (Mountford and 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 ranging from novices to experts.

Acknowledgements

We gratefully acknowledge the support to our laboratory from the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario, the Institute for Robotics and Intelligent Systems of Canada, Apple Computer's Human Interface Group and Advanced Technology Group, Xerox PARC and EuroPARC, Digital Equipment Corporation, the IBM Canada Lab Centre for Advanced Studies, and Alias Research. We are indebted to Marc Eisenstadt and John Domingue who provided valuable comments on a draft of this paper.

References

Albright, L., J.M. Francioni and J.A. Jackson (1991). Auralization of Parallel Programs. Presentation at CHI '91.

Baecker, R.M. (1981). *Sorting Out Sorting*. Dynamic Graphics Project, University of Toronto. 16 mm colour sound film/videotape, 30 minutes, presented at ACM SIGGRAPH '81 (distributed by Morgan Kaufmann, Los Altos, CA).

Baecker, R.M. and J.W. Buchanan (1990). A Programmer's Interface: A Visually Enhanced and Animated Programming Environment. *Proceedings of HICSS '90*, 531-540.

Baecker, R.M. and W.A.S. Buxton (1987). *Readings in Human-Computer Interaction*. Reading, MA: Addison-Wesley.

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

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

Baecker, R.M., I.S. Small and R. Mander (1991). Bringing Icons to Life. *Proceedings of CHI '91*, 1-6.

Bentley, J.L. and B.W. Kernighan (1981). A System for Algorithm Animation: Tutorial and User Manual. Computing Science Tech Report 132, AT&T Bell Labs, January 1987.

Brooks, F.P. (1989). University of North Carolina (Chapel Hill) Research. In Implementing and Interacting with Real-Time Microworlds, Course #29, ACM SIGGRAPH '89.

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

Brown, M.H. and J. Hershberger (1991). New Techniques for Algorithm Animation. In *Proceedings of The IEEE Workshop on Visual Languages*, IEEE Computer Society.

Brown, M.H. and R. Sedgewick (1984). A System for Algorithm Animation. *Computer Graphics* 18(3), 177-186.

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

Fisher, S.S. (1990). Virtual Interface Environments. In *The Art of Human-Computer Interface Design*, B. Laurel (Ed.), Reading, MA: Addison-Wesley, 423-438.

Goldstein, H.H. and J. von Neumann (1947). Planning and Coding Problems for an Electronic Computing Instrument. Reprinted in *von Neumann, J., Collected Works*., A.H. Taub. (Ed.) New York: McMillan, 80-151.

Haibt, L.M. (1959). A Program to Draw Multi-Level Flow Charts. *Proc. of The Western Joint Computer Conf.*., 131-157.

Conventional Wisdom:
Software Visualization for Software Engineering

The old CW on program visualization for software engineers is that it's a great idea that never quite measures up to the expectations. By following the recipe given below, the new CW is likely to be vastly improved.

CHARACTERISTICS		Conventional Wisdom
Scalability	⇧	Real programmers work with real programs; PV should too.
Multiple Programs	⇔	Two versions possible, but not an everyday occurrence.
Data Visualization	⇧	Programs are the story of data that changes.
Colour & Animation	⇧	Black and white stills may be arty, but PV needs serious bandwidth.
Sound	⇧	Imagine Madonna without the music. Well, maybe not.
Modified Source	⇩	You really want to do this to a million-line program?
Live	⇧	Who wants to see yesterday's news? CNN for the 90's.
Customizable	⇧	When did the contractors ever get it right the first time?
Code Familiarity	⇩	If you have to know it to see it, what's the point?
Invasive	⇩	This is real code we're talking about; beware of quicksand.
Procedural Customization	⇔	It's where the real power is, but what a pain.
Same Language	⇔	A nice idea, but it tends to clash with unrestricted program class.
Interactive & Declarative Customization	⇧	Old CW: Too hard. New CW: It's the way of the future.
Navigation & Elision	⇧	Real programs are big, and it's easy to get lost.
Experimental Evaluation & Production Use	⇧	Why should I believe that your system is going to help me?

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

McCormick, B.H., T.A. DeFanti, and M.D. Brown (Eds.) (1987). Visualization in Scientific Computing. *Computer Graphics*. 21(6), special issue.

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

Myers, B.A. (1983). Incense: A System for Displaying Data Structures. *Computer Graphics* 17(4), 115-125.

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

Rosenblum, L.J. and G.M. Nielson (Eds.) (1991). Visualization. *IEEE Computer Graphics and Applications* 11(3), special issue.

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

Small, I.S., R.M. Baecker and B.A. Price (in preparation). Software Visualization.

Sonnenwald, D.H., B. Gopinath, G.O. Haberman, W.M. Keese III and J.S. Myers (1990). InfoSound: An Audio Aid to Program Comprehension. *Proceedings of HICSS '90*, 541-546.