Report Number: WUCS-92-06

1992

Program Visualization: The Art of Mapping Programs to Pictures

Gruia-Catalin Roman and Kenneth C. Cox

In this paper program visualization is defined as a mapping from programs to graphical representations. Simple forms of program visualization are frequently encountered in software engineering. For this reason current advances in program visualization are likely to influence future developments concerning software engineering tools and environments. This paper provides a new taxonomy of program visualization research. The proposed taxonomy becomes the vehicle through which we carry out a systematic review of current systems, techniques, trends, and ideas in program visualization.

School of Engineering & Applied Science

Program Visualization: The Art of Mapping Programs to Pictures

Gruia-Catalin Roman
Kenneth C. Cox

WUCS-92-06

February 1992

This paper appeared in *Proceedings of the 14th International Conference on Software Engineering*, May 1992, pp. 412-420.

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Program Visualization: The Art of Mapping Programs to Pictures

Gruia-Catalin Roman and Kenneth C. Cox

Department of Computer Science
Washington University
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130

Abstract

In this paper program visualization is defined as a mapping from programs to graphical representations. Simple forms of program visualization are frequently encountered in software engineering. For this reason current advances in program visualization are likely to influence future developments concerning software engineering tools and environments. This paper provides a new taxonomy of program visualization research. The proposed taxonomy becomes the vehicle through which we carry out a systematic review of current systems, techniques, trends, and ideas in program visualization.

Every science begins as philosophy and ends as art.
W. J. Durant
(American educator)

1. Introduction

Computers are an integral part of our communication-centered society. Networking, desktop publishing, and electronic mail are only three examples involving computers in service of communication. With the advent of multi-media communications, computers have begun to support increasingly sophisticated forms of human interactions. The generation, transmission, interpretation, and use of pictures is becoming a routine activity. Cost and performance are reaching acceptable levels even for compute- and storage-intensive applications such as animation, three-dimensional imagery, and live video. These developments led to the emergence of a new area of research in computing: *visualization*—a field of study concerned with the use of graphical representations in the computing milieu.

Advocates of visualization point to the important role imagery plays in human communication in general, to the extraordinarily high bandwidth of human visual system, to the speed with which humans track and detect visual patterns, to the expressive potential of a vocabulary that exploits multiple dimensions, and to the power of abstraction inherent in pictorial representations. Successful use of graphical elements in computer-related endeavors such as human interfaces and scientific visualization provide credible evidence of the extraordinary potential of visual communication.

Two visualization subfields, visual programming and program visualization, are of particular import to software engineering. *Visual programming* is concerned with graphical specification of computer programs, while *program visualization* deals with graphical presentation, monitoring and exploration of programs expressed in textual form. Primitive forms of visual programming and program visualization have been part of the software engineering culture from its inception and recent developments in the visualization field were received with renewed enthusiasm. Unfortunately, in software engineering graphical representations have often been viewed as a panacea for a variety of problems relating to technical communication. Exaggerated claims and unreasonable expectations are commonplace in both academic papers and marketing brochures. Pictures are often used in situations where text would be more economical; precise, smooth integration of diagrams and text is rare; and the lack of a strong formal framework is sometimes glossed over by appeals to the so-called intuitive nature of the graphics.

Mere faith in the worth of pictures is not sufficient to make them an effective communication vehicle. As far as software engineering is concerned, visualization is called upon to open up new avenues of communication. It is not the case that we are attempting to provide computer support for something that is well understood, as was the case with desktop publishing. We are seeking to produce a world in which computing and visual communication are merged and harmonized. While visual programming and program visualization do not yet hold all the answers we seek, they offer valuable inspiration, interesting new ideas, and, possibly, a glimpse to the future of CASE technology.

Several books and survey articles in print provide extensive coverage of visual programming but only limited treatment of program visualization. This paper attempts to remedy this situation by focusing exclusively on the state of the art in program visualization. The presentation is organized around a new taxonomy of program visualization. A formal definition of program visualization as a mapping from

This work was supported in part by the National Science Foundation under the Grant CCR-9015677. The Government has certain rights in this material.

programs to graphical representations (Section 2) supplies an objective technical justification for the taxonomy proposed in this paper. Section 3 considers what aspects of a program one may want to visualize. Section 4 organizes the types of mappings encountered in program visualization from the point of view of their power of abstraction. Section 5 focuses on methods used to specify or construct the mapping from programs to graphical representations. Section 6 explores visual presentation techniques, ranging from pedagogical concerns to the use of color and animation styles.

2. A Taxonomy of Program Visualization

Developing a taxonomy is always a difficult task. The challenge is to select classification principles that differentiate cleanly among recognized contributions to the field, offer meaningful insights into the workings of the various systems, and help identify likely prospects for future developments. Each published survey to date employs a different taxonomy. Shu [29] focused on increasing degrees of sophistication exhibited by program visualization systems, from pretty-printing to complex algorithm animations. Myers [23] used a classification along two axes: the *aspect* of the program that is illustrated (code, data, algorithm) and the *display style* (static or dynamic). Chang [8], although not providing a taxonomy, characterized program visualization as the use of visual representations to illustrate program, data, the structure of a system, or the dynamic behavior of a system. Finally, Brown [3] proposed classifying algorithm animations along three axes: *content* (direct or synthetic representation of information about the program), *transformation* (discretely or smoothly changing images), and *persistence* (representations of the current state or of the entire execution history). Although each of these taxonomies has its own rationale and merits, we find them less than satisfactory because they are not based on a well-formulated model or theory of the field. Our search for an adequate (if not necessarily definitive) model of program visualization starts with an attempt to clarify what we mean by the term.

We see visualization as a mapping from programs to graphical representations. In this light, the meaning of the term program, the available graphical vocabulary, the kinds of mappings being contemplated, and the means to construct these mappings become the defining features of a program visualization system. This definition also suggests that the process of visualization is the result of an interplay among three participants: the *programmer* who develops the original program; the *animator* who defines and constructs the mapping; and the *viewer* who observes the graphical representation. While, these are only stylized roles meant to help us organize and present the material, the specialized expertise required of each role may actually lead in practice to a natural division of labor among distinct individuals.

As intimated above, this particular perspective on visualization leads to a classification of program visualization systems centered around characterizations of the domain, the range, and the nature of the mapping they support. We identify below four axes along which we classify program visualization systems. The first and the fourth relate to the domain and the range of the mapping, respectively. The other two deal directly with the mapping. Figure 1 illustrates our model of

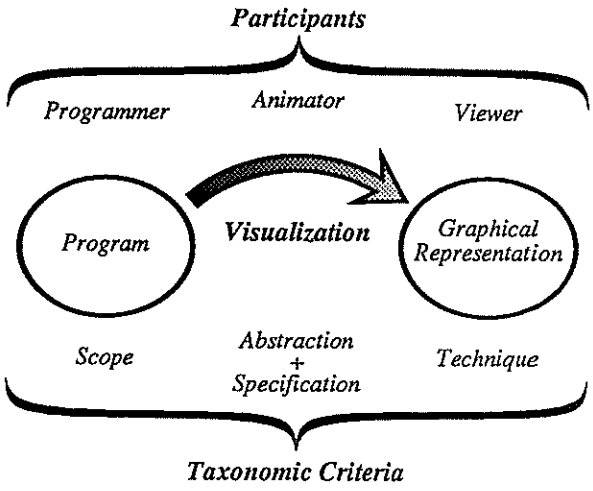

Figure 1. Visualization as a mapping from programs to graphical representations.

visualization and clarifies the taxonomic criteria and the roles of the participants.

(1) *Scope*—what aspect of the program is visualized? The domain of an individual visualization is a program. Formally, a program can be characterized by its code, by its data and control states, and by its execution behavior. Visualization systems often limit their scope to a subset of these program aspects. A similar criterion was suggested by Myers and Chang.

(2) *Abstraction*—what kind of information is conveyed by the visualization? This criteria relates both to Shu's classification with respect to degree of sophistication and to Brown's notion of content. The issue is the level of abstraction associated with the concepts presented in a graphical form by the particular system. Highlighted code, for instance, offers a very low-level representation of the control state. An animation, on the other hand, augments a behavior depiction with carefully choreographed explanatory information.

(3) *Specification Method*—how is the visualization constructed? This issue, ignored by most published surveys, is essential to understanding the power and flexibility of a particular system. Some systems provide "hardwired" mappings, while others may allow for arbitrary redefinitions of the mapping; some focus on mapping program states while others focus on events; some require modification of the code while others do not.

(4) *Technique*—how is the graphical representation used to convey the information? This criterion considers issues having to do with effective visual communication. They include the visual vocabulary, the use of specific visual elements to convey particular kinds of information, the organization of visual information, and even the order in which material is presented to the viewer.

In the next four sections we further refine each of these classification criteria, highlight representative research efforts, and attempt to uncover emerging trends.

3. Scope

The purpose of program visualization is to extract information about some aspect of a program and present it in a graphical form. Precisely which aspects of the program are examined serves as a major defining characteristic of visualization systems. Although the trend is toward the ability to visualize all aspects of a program and particularly its execution, different systems may still emphasize one aspect over the others. This is certainly true if we take a historical perspective. *Code*, in the sense of the text of the program, has been a primary area of interest from the beginning and remains important. Visualization of the *data state* was also an early interest, and most program visualization systems still focus on presentation of the internal data of the program. *Control state* gained special importance in sequential programs exhibiting recursion and backtracking and, more recently, in concurrent programming. Finally, the concern with program *behavior* started with input/output monitoring and, with advances in computing power and program visualization methods, emerged as the focal point of some of the most sophisticated program animations.

Code. Pretty-printers represent the most primitive form of code-oriented program visualization. They are concerned with increasing code readability by transforming syntactic patterns into page layouts. With the advent of laser printers, programs that exploit multiple fonts, shading, and simple line graphics to produce attractive code listings have become available.

Programs to produce graphical presentation of code have been available for years; one of the earliest systems [14] produced flowcharts. Initially these programs were rather limited, producing statement-level diagrams of the program's structure. Following a trend toward greater abstractive power, modern systems permit a single program to be viewed in several different ways, from tree structures of the expressions within a statement to block diagrams of the interconnections between program modules. These are of course the visual CASE systems, of which Garden [25] and PegaSys [21] may be taken as representative examples. Many of these systems are both program visualization systems and visual programming systems, permitting the user both to view existing code and to construct new code by appropriate arrangement of graphical elements.

Data state. Early data-visualization systems were hampered by the relatively slow processors available, necessitating somewhat oblique methods. Two early algorithm animations [1, 17] were recorded on film frame-by-frame, with each frame representing the results of a single state snapshot. The visualizations in these films use a relatively low level of abstraction, typically extracting only scalar variables such as the values held in an array. Increasing processor speeds permitted correspondingly more powerful monitoring and abstraction techniques. These have been exploited in several debuggers (e.g., Aktri [24] and Incense [22]) for the examination of complex data structures. Database systems that permit graphical representation of database queries and contents are close relatives of this type of visualization system.

The most recent program visualization systems permit smooth animations of complex data structure representations. Examples include ALADDIN [15, 16], ANIMUS and its relations [10, 19], BALSA [4, 7], PAVANE [26, 27], PROVIDE [20], and TANGO [30, 31]. This list is not comprehensive, but each of these systems has one or more points of interest that will be discussed in later sections.

Control state. A sequential program's control state is, in a sense, represented by the processor's program counter, but few systems use such a low level of abstraction. Instead, higher-level constructs such as statements and procedures are presented. Historical information, such as calling sequences, is often additionally maintained. In BALSA, for instance, the currently-executing portion of the code can be presented in a window. Other systems relate control information to the overall structure of the program, for example by displaying the code structure as a module-interconnection diagram and highlighting the module currently having control.

Several program visualization systems have been designed for use with languages and models whose control state is more complex. One such system is TPM [12], the Transparent Prolog Machine, which visualizes the goal-directed behavior of Prolog programs. Goals are presented as tree structures that evolve as the goal is processed; unification, backtracking, and the behavior of such special operations as the cut are graphically rendered. One system that attempts to demonstrate the behavior of object-oriented programs in the same way that TPM attempts to present Prolog behavior is the Object-Oriented Diagramming system [9], which displays the message-passing behavior of objects. Other systems use an object-oriented framework to construct visualizations. Generally in these systems objects have a method that transforms the object state into some graphical information. The ANIMUS system developed by Duisberg *et al* is an example of this type of system. The ParVis system [18] provides low-level visualizations of the control state in parallel Lisp programs, displaying the generation and progress of subtasks through time.

Behavior. The simplest (and oldest) means of observing a program's behavior is by examining its input/output actions. This technique can be used to provide a meaningful abstract treatment of the program: it is simply considered to be a "black box" that transforms input to output. This approach has important theoretical implications, but gives little insight into the algorithm and is not applicable to all programs of interest. A more detailed view of the program's behavior, based on its internal activities, is needed for intuitive understanding.

Every program may be viewed as performing a series of atomic transformations of its state. Observation of these transformations, formally known as events, permits development of an understanding of how the program accomplishes its function. Many program visualization systems attempt to capture and present this kind of information. There are differences, however, in the granularity and the range of the events they consider. The granularity can vary from primitive machine instructions to large operations

performed by entire blocks of statements. The events of interest may be changes in the values of specific variables, entry and exit from procedures, or communication activities. While some visualization systems predefine the events of interest, most allow the animator to define which events are meaningful for a particular visualization; for details, refer to the next section.

With the advent of distribution and concurrency, there is a growing need to recognize and visualize events that are the result of independent state changes at multiple sites. This often occurs in the area of system monitoring, and many investigators are turning to using program visualization techniques to present the data they are collecting. The difficulties of using operational thinking with concurrent programs has led some researchers to consider the issue of visualizing not the operational details of the program but the abstract properties used in formal reasoning about concurrent computations.

4. Abstraction

In this section we consider the information that is conveyed via the graphical representation. We discriminate among systems based on the level of abstraction supported by the visualization system. Abstract representations of the program are needed to control complexity during exploration and monitoring and to facilitate program understanding in a pedagogical setting. Display size limitations are another factor requiring the use of compact abstract representations. Our taxonomy distinguishes among five levels of abstraction, illustrated in Figure 2. We consider each in turn from the least to the most abstract representation of the program. The boundaries between the various levels are imprecise, and in practice a visualization system is likely to support multiple levels of abstraction, separately and in combinations. For this reason we cite specific systems only when they offer some unique capabilities, or to illustrate a particular point.

Direct representation. The most primitive graphical representations are obtained by mapping some aspect of the program directly to a picture. Because only very limited abstraction mechanisms are employed, it is often the case that the original information can be easily reconstructed from the graphical representation. Formatted layouts of the code, gauges set to indicate the values of variables, two-dimensional representations of linked lists and binary trees, and color encodings of values stored in an array are some examples of direct representations. Other forms are encountered primarily in CASE and debugging systems: flowcharting and similar graphical representations of code structure, monitoring of control flow through display of the currently-executing statement, and tracking of procedure invocation by presenting the call stack (possibly as overlapping windows, one per invocation).

Structural representation. More abstract representations may be obtained by concealing or encapsulating some of the details associated with the program or its execution and using a direct representation of the remaining information. Two- or three-dimensional diagrams and graphs are typically used to depict program structures (e.g., structure charts), network connectivity, and data access capabilities; in each case a complex object composed of a number of subcomponents is treated as a single simple object, with its internal structure hidden. Many other examples can be cited. Histograms may encode the relative frequency of message occurrences by type, concealing other message details. In an operating system, proportionally-sized colored blocks may indicate memory allocation and usage without attempting to represent the state of the memory. In all these cases, the information presented to the viewer is present in the program, although possibly obscured by details. The representation simply conveys the information in a more economical way by suppressing aspects not relevant to the viewer.

Synthesized representation. Representations in this category are distinct from structural representations in that the information of interest is not directly represented in the program but can be derived from the program data. This shift of perspective often occurs when the data representations selected by the programmer come into conflict with the needs of the animator. The animator may prefer to emphasize other aspects of the algorithm that, although logically maintained by the program, have no explicit representation. The animator must then construct and maintain a representation that is more convenient for his particular needs. As an example, an animator might choose to stress the work that remains to be done, while the programmer might prefer to represent the work that has already been done—or vice-versa.

Although many systems provide for forms of synthesized representations (if only in a limited fashion), few animations fully exploit the possibilities. This situation may be partly attributed to the fact that few systems have the ability to construct arbitrary mappings from programs to pictures. The relative newness of the field may also play a part—even animations produced by BALSA, one of the older systems, involve only limited use of synthesized representations. In BALSA, the animator can construct synthesized representations by defining data structures that are shared by the graphical procedures. These procedures (also written by the animator) can then operate using both data exracted from the program and the contents of the shared data structures.

Analytical representation. These are representations that attempt to capture highly abstract program properties. The idea is to de-emphasize the operational mechanics of program execution and focus on issues that are important in analytical thinking about the program such as correctness. One example might be in a sorting algorithm, where rather than the usual direct representation of the elements that are being sorted the animator might instead attempt to capture an invariant such as *"All elements with index less than the value of variable k are correctly sorted"*. Considering an algorithm in terms of its properties often leads to new and interesting animations which would probably not be developed using more traditional methodologies.

This approach is rooted in the notion that properties that are important in reasoning formally about programs should, when translated into visual form, also help the viewer understand the program behavior. So far, most work in this area has centered around PAVANE, which attempts to provide a framework for visualization of concurrent computations—a domain in which formal reasoning about program correctness is far more crucial than in the sequential arena, where

(a)
```
AllPairs( integer N, real array A[0..N-1,0..N-1] ) begin
    integer k, i, j
    for k := 0 to N-1 begin
        forall i := 0 to N-1, j := 0 to N-1 parbegin
            A[i,j] := min( A[i,j], A[i,k] + A[k,j] )
        parend
    end
end
```

Figure 2(a). A parallel version of the Floyd-Warshall algorithm for finding the shortest distance between all pairs of nodes in a graph. The graph is supplied in the form of an array *A* where *A[i,j]* is the distance from node *i* to node *j* if an edge connects the two nodes and is infinity otherwise. At each step of the algorithm, the index *k* is scanned and all entries of the array are updated in parallel.

(b)

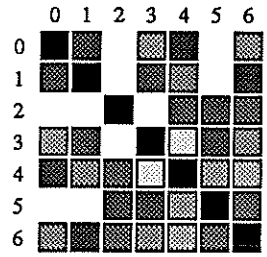

Figure 2(b). A direct representation of the data state of the program after several steps (specifically, the code has been executed for *k* equal to 0, 1, and 2). The contents of array *A* are represented using rectangles whose color (here, grey tones) encode the array values, with black representing a value of zero. If an array entry contains the value infinity, the rectangle is omitted.

(c)

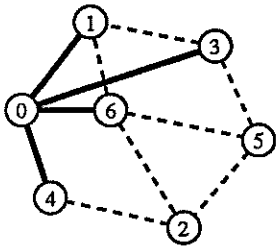

Figure 2(c). A structural representation of the same data. Those distances from node 0 which are known are indicated by heavy links; this is superimposed on a representation of the initial graph drawn with dotted lines. Again only the fact that the distance is known is represented. This is an illustration of *information concealment*, where some details are hidden from the user; this diagram may be seen as simply a "slice" through the first row of Figure 2(b).

(d)

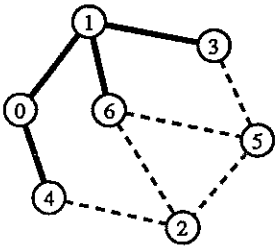

Figure 2(d). A synthesized representation. A current shortest path from node 0 to each other node is drawn. This information is not represented in the program but can be extracted by the visualization system. As was the case in Figure 2(c), only the fact that the distance and path is known is shown, not the actual distance. Also note that any node could be used in place of node 0; one can imagine a system where the viewer could select any of the nodes of the graph and immediately see the corresponding paths.

(e)

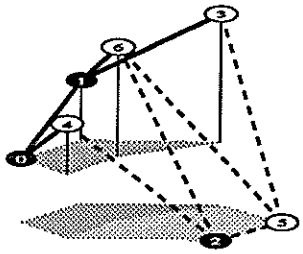

Figure 2(e). An analytical representation which attempts to illustrate the invariant *"If the distance from node i to node j is known, all internal nodes on the path have been scanned."* This is shown by coloring the scanned nodes, allowing the viewer to easily perceive the property. This representation also uses the Z-coordinate to encode distances, placing node 0 at a height of 0, nodes of known distance at the appropriate Z-coordinate, and nodes of unknown (infinite) distance at a small negative Z-coordinate—a somewhat arbitrary choice.

(f)

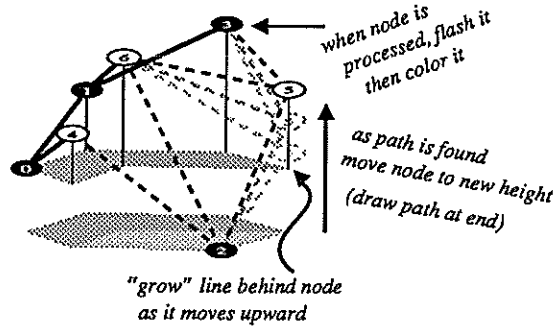

Figure 2(f). Here we attempt to convey the essence of an explanatory presentation, one which uses extra visual events to direct the viewer's attention. Naturally the limitations of the print medium prevent a full illustration, but we hope this figure illustrates the concept. In the algorithm, index 3 has just been scanned and as a result the distance/path from 0 to 5 is now known. The viewer's attention is first directed to the scanning by flashing node 3. The fact that the path to 5 is now known is illustrated by animating the repositioning of node 5; this is visually clearer than having node 5 "leap" in a discrete step from one position to the other. Note that while node 5 moves, none of the images represent any actual state of the computation.

Figure 2: Illustration of the various levels of abstraction possible in a visualization.

operational presentations of algorithm behavior are often sufficient.

Explanatory representation. Sophisticated visualizations go beyond presenting simple representations of the program state and use a variety of visual techniques to illustrate program behavior. These visual events often have no counterparts in the computation being depicted by the animation. They are added for the sake of improving the aesthetical quality of the presentation, out of the desire to communicate the implications of a particular computational event, and in order to focus the viewer's attention. In essence, the animator takes the liberty of adding events to the representation of the program in order to "tell the story" better. Again a sorting algorithm provides a simple example. Two rectangles may smoothly swap positions to indicate the exchange of two array elements; the intermediate images while the rectangles are moving do not represent any actual states of the computation.

5. Specification Method

Whether a visualization system maps syntactic units, program states, or events to images, the key to acceptance is the ease with which the animator can construct the kinds of visualizations needed for the task at hand. The exploratory activities involved in testing and debugging demand high degrees of flexibility in defining and redefining the mapping, while execution monitoring requires minimal intrusion in the computation process. In this section we discuss the way different program visualization systems approach the task of specifying visualizations.

Predefinition. Application-specific visualizations often employ a fixed or highly constrained mapping. The animator has little or no control over what is visualized and in what way the information is presented. Most CASE systems opt for this approach because their intent is to support and enforce a particular methodology and graphical notation. Although the expressiveness of these systems is often limited, they have the advantage of speed; the constraints permit the use of specialized visualization algorithms that have been optimized for performance.

Annotation. This approach was pioneered by BALSA and gained widespread acceptance in the algorithm animation field. The animator augments the program with calls to procedures that allow the animator to construct and modify images. The placement of these calls (annotations) corresponds to the occurrence within the code of events deemed significant to the behavior of the algorithm. Information about the program state is passed via the parameters of the invoked animation procedures. In this manner computational events are mapped to arbitrary sequences of visual events, i.e., changes in the graphical representation. TANGO uses a similar approach but emphasizes the production of smooth animations. TANGO visualizations are treated as a mapping from program events to animation actions. The occurrence of an event of interest triggers some smooth animation of one or more graphical objects in the image.

The annotative approach has several advantages, chief of which is the animator's ability to define events at any suitable level. For example, a sorting algorithm might be animated by detecting each comparison and swap; however, if the same sorting algorithm were used as a subroutine in another algorithm, the animator might choose to present only the final result. The ability to write application-specific animation procedures to handle each event is both an advantage and a disadvantage, in that it permits versatility but requires additional work; this can be somewhat mitigated through the use of libraries of routines. The most significant shortcoming of the method is the need to access and modify the code of the program.

Association. A less intrusive alternative is to allow the user to define a relationship between the program state and the attributes of graphical objects or icons. Each change to the selected portion of the state (typically data values) causes a corresponding change in the image. This is the approach used by PROVIDE, which allows the animator to directly map the values of variables to attributes of icons such as gauges and slider bars. PROVIDE also permits the viewer to interact with the computation by manipulating the icons, thereby changing the corresponding program values. PVS [13] uses a similar approach, but is intended for monitoring real-world processes (e.g., a power plant) rather than programs. The Garden visual programming system permits the programmer to define *visual* data types whose attributes are a part of the program state, an interesting variation of the basic associative idea.

The chief advantage of the associative approach—the automatic updating of the image in response to changes in the state—is also one of its main disadvantages, since it is difficult to capture state changes at an appropriate level. The problem typically arises when several primitive state changes should be considered as a single logical change (for example, when two variables are exchanged using three assignment statements); the image should not be updated to reflect the intermediate results. Most systems that use associative visualization provide mechanisms to indicate (generally by annotation) when the image should be updated. Another disadvantage is the rather low level of abstraction enforced by the mapping of simple variables to graphical attributes.

Declaration. The declarative approach resembles the associative approach in that the animator specifies a mapping between the program state and the final image, and changes in the state are immediately reflected in the image. However, the declarative approach permits arbitrarily-complex mappings instead of the simple one-to-one mappings of the associative approach. Thus, arbitrary predicates over the state can be expressed and visually captured. PAVANE is an example of this type of system. PAVANE models the program state as a set of tuples and uses a rule-like notation to specify a mapping between the state and the image; the mapping may be composed of several sub-mappings. ALADDIN also uses a declarative approach to define the relationship between program variables and the image, but combines this with an annotative approach which indicates at what points in the program the image should be updated; the annotations are specified graphically. ANIMUS and its parent system ThingLab [2] use the declarative approach in two ways. These are object-oriented systems, and each object can have a graphical representation that is automatically updated in response to changes in the object. More significantly, the animator can declaratively specify *constraints* on the relations between objects, and the system will ensure that these

constraints are maintained (e.g., by moving the representation of an object).

The greatest advantage of the declarative approach is its abstractive capability, but this increased power also requires more processing to map the program to the final image. In addition, manipulation of the program by the viewer (through modifications of the graphical objects) is more difficult than in the associative case since a one-to-one relation between variables and object attributes does not exist. The developers of PAVANE have recently begun exploration of the use of "reverse rules" to map viewer interactions with the image onto changes in the program state.

Manipulation. Systems that use manipulation (also called *animation by demonstration*) specify visualizations through the use of examples. The gestural system developed by Duisberg [11] is one of the few program visualization systems to use this approach. This system attempts to capture the gestures used by the animator as she directly manipulates an image and to tie these gestures to specific program events. Duisberg illustrated his system with a sorting algorithm, where the exchanges of array elements were animated by defining a "swap gesture" that exchanged the positions of two rectangles. This gesture was then tied to the exchange event in the sorting algorithm by selecting the appropriate portion of the program code. Although exciting in principle, specification by manipulation suffers from the difficulty of specifying the exact relationship between the gesture and the program event. This can be accomplished for specific cases, but a general approach has yet to be defined. Investigation of a mixed approach, where *motions* are specified gesturally but the binding between events and some attribute of the motion is specified in a more traditional matter, would undoubtedly prove fruitful. Stasko has recently described [31] such a gesture-capturing system for use with TANGO.

6. Technique

In this section we examine techniques for visual representation of information about programs. We focus on approaches that are either fundamental or that have been tried and appear to enhance visual communication. The presentation is slightly biased towards algorithm animation (i.e., presentation rather than monitoring and exploration) because this is the area in which the most avant garde techniques are being explored. Brown recently produced a related survey [6] of program visualization techniques with illustrations from work on BALSA and Zeus [5].

Program visualization is such a young field that no definitive lexicon of techniques exists. In addition, new techniques (such as the integration of sound) continue to be added. For this reason, although we try to give a reasonably complete coverage of the area, we make no claims of being comprehensive. The discussion is organized around four topics: selection of sample executions to be visualized; design of the basic screen organization; visual encoding of information; and enhancements of the basic visual presentation.

Sample execution selection. The complexity of a visual presentation is generally proportional to the amount of information being conveyed. Therefore, it is advisable to launch a presentation with a relatively small problem instance and gradually introduce larger ones as the viewer begins to understand the meaning associated with the visual patterns unfolding on the screen. For pedagogical purposes one may want to consider pathological cases, such as providing already-sorted or inversely-sorted arrays to a sorter, or to bias the selection of data, for example by guaranteeing that all elements of an array are distinct. When exhibiting the behavior of nondeterministic algorithms, the ability to influence the choices made by the program is useful, as is the ability to present the results for each possible choice side-by-side. A similar technique is to present the history of selected aspects of the computation, either in separate windows or incorporated into the primary image. This allows current activities to be readily understood in the broader context of previous execution steps. All these techniques have been shown to provide invaluable assistance in building the viewer's confidence and intuition; the comprehensive library of BALSA animations of sorting and path algorithms provides one example of the effectiveness of these techniques.

Screen design. This category of issues refers to the overall structure of the final image, including placement on the screen and internal characteristics of the graphical model. The most basic decision is the dimensionality of the model. Early systems were restricted to two-dimensional models, but three-dimensional models are now in use. Certain specialized applications of scientific visualization use spaces of high dimensionality which are projected onto a three-dimensional space for display. As might be imagined, additional dimensions can be effectively used to enhance data representations. One example can be seen in the video tape *Diffusing Computations* [28], which presents a termination-detection algorithm using three dimensions.

The next factor to be considered is the placement of the *viewpoint* (the location of the viewer's eye in model space). In a few systems this is at a fixed location, but most systems permit the viewer to move the viewpoint. In two-dimensional systems, where the viewpoint is generally treated as being directed "down" from some distance "above" the plane in which the objects are placed, viewpoint motions are limited to simple scrolling of the image and movement of the viewpoint closer to or farther from the image plane. The situation is more complex in three dimensions, since the direction of the view must also be modified. Controls that permit the viewer to "fly" through a three-dimensional layout have been used in scientific visualization. PAVANE places the viewpoint using a polar coordinate system and allows certain simple transformations of the viewpoint, e.g., a slow rotation of the entire model, to be automatically performed.

Several systems permit multiple views of a single algorithm and views of several different algorithms to be displayed simultaneously, generally in different windows (and with separate viewpoint controls). The first technique is particularly useful when experimenting with different representations to determine which is most effective, but is also useful in the final presentation because the viewer can focus on the representation that she finds most effective. The second technique allows the performance of several algorithms to be contrasted; BALSA has used this to good effect in the presentation of sorting and bin-packing algorithms. With both techniques the ability to compare two images leads to

greater insight into the behavior of the program or programs that are represented.

Information encoding. Tufte [32] has produced an excellent general survey of information encoding. In the broadest terms, the means for visual expression at the disposal of the animator fall into three categories: visual objects and their attributes, visual relationships among such objects, and visual events.

Visual objects are generally used to encode relatively concrete aspects of the program, such as the values of individual variables or the identity of enabled actions. The simplest visual objects are abstract geometric entities such as points, lines, rectangles, circles, and spheres. Virtually all program visualization systems provide such primitive objects. Their use has been highly effective and they remain a favorite of many animators because they can be combined in arbitrary ways, have a variety of properties that can be used to encode values, and carry no predefined semantic content. Simple geometric entities may be combined to form complex objects or objects with multiple moving parts. Complex combinations can be used to model physical processes or machines. A variant of this approach—the "widget"—is widely used. A widget is basically a pre-defined complex object that may be treated as a simple object having certain properties; a slider gauge is a common example of a widget, but more complex examples such as pie charts and two-dimensional data plots are also used. A third kind of visual object is the icon, defined as a graphical entity with a fixed semantic interpretation and (generally) fixed syntactic rules for its use. Icons are very popular with visual language designers but not widely used in animations.

One significant shortcoming of most program visualization systems is the lack of facilities to define new visual objects. ANIMUS provides a system whereby new objects can be defined as collections of one or more objects. The properties of the new object can be accessed by the component objects, and additional constraints can be placed on the relationships between the member objects. This approach would undoubtedly prove useful if incorporated in other systems.

In a typical visualization some of the information being conveyed is represented by the attributes of the objects (color, size, and so forth), but as much or more information is conveyed by the visual relationships between objects. This remains very much an art, but certain general principles can be identified. Structural properties of the program can be captured by analogous geometric structures, e.g., a two-dimensional array may be represented by a tiled rectangle, or processes that execute in a round-robin fashion may be arranged in a ring. More abstract properties may be visualized by enforcing particular geometric alignments and coloring rules. For instance, the fact that in a bin-packing algorithm the element that is currently being packed does not fit in certain of the bins is a program invariant that may be easily observed by representing both bins and elements by rectangles whose length is proportional to the bin capacity or element size and lining up the rectangles so the lengths can be visually compared. Despite the importance of layout, few systems provide explicit means to enforce these restrictions, relying instead on implicit information. For example, an animation might implicitly represent the layout of the elements of an array by using a function that converts the indices of the array elements to the X-coordinates of the objects representing the array. ANIMUS is one of the few systems that permits explicit specification of layout through the use of constraints on the geometric relationships between elements.

Visual events involve both discrete and continuous changes in the graphical representation. Discrete transitions are generally used when trying to convey an intuition about the overall behavior of a program in a context where the changes can be easily grasped; BALSA's visualization of the QuickSort, in which each discrete change shows the results after each partitioning step, may be cited as an example. Continuous transitions are preferable to illustrate small steps and when the nature of the state change must be explained to the viewer. Smooth changes also permit easy discrimination between attributes that change and those that remain unaffected—a nicety that might not be clear from a discrete change. The TANGO system emphasizes the importance of such smooth animations.

Presentation enhancements. Special effects are often used to improve the visual quality of the presentation. The use of smooth reflective surfaces, for instance, has powerful aesthetic appeal. Other enhancements, however, are both pleasing and convey semantic information. The slow evolution of an object from a point to its final size while it also moves along some predefined trajectory is not only visually attractive; it can give the viewer time to process the information and may also be used to define a relationship between the context appearing at the point of origin and the new object being introduced. Highlighting particular objects through color changes or transparent color overlays serves to focus the viewer's attention on those objects that are the protagonists of the current state transition. A similar technique is the use of color shifts to encode the "age" of objects, which can both focus attention on the most recent changes and enhance the perception of historical trends in algorithm behavior.

7. Conclusions

In this paper we introduced a new taxonomy for program visualization, one derived from and motivated by our view of program visualization as a mapping from programs to graphical representations. We used this taxonomy as a guide to surveying current thinking and techniques in program visualization. Our hope was to provide a broad and coherent picture of this very active and exciting field. As program visualization matures, a natural sorting out of the various techniques is bound to take place and stereotypical uses of particular approaches will get established. At that time program visualization may become a fairly routine activity with well-defined principles adhered to by most and occasionally questioned and revised by few adventurous souls. Today, however, it is an exciting area where experimentation, innovation, and creativity reign supreme.

Acknowledgments: We would like to thank K. Goldman, T. D. Kimura, R. F. Gamble, J. Y. Plun, B. Swaminathan, and C. D. Wilcox for their constructive criticisms regarding this paper.

References

Several of the papers referenced here can be found in the two-volume *Visual Programming Environments* collection edited by E. P. Glinert and published by the IEEE Computer Society Press. The collection focuses on visual programming but is an excellent sourcebook for anyone interested in the field of visualization.

[1] Baecker, R. M., *Sorting Out Sorting* (film), Dynamic Graphics Project, University of Toronto, Toronto, Canada, 1981.

[2] Borning, A., "The Programming Language Aspects of ThingLab, a Constraint Oriented Simulation Laboratory," *ACM Transactions on Programming Languages and Systems*, vol. 3, no. 4, pp. 353-387, 1981.

[3] Brown, M., "Perspectives on Algorithm Animation," *CHI'88 Human Factors in Computing Systems*, Washington, DC, USA, pp. 33-38, 1988.

[4] Brown, M. H., "Exploring Algorithms using Balsa-II," *IEEE Computer*, vol. 21, no. 5, pp. 14-36, 1988.

[5] Brown, M. H., "Zeus: A System for Algorithm Animation and Multi-View Editing," *1991 IEEE Workshop on Visual Languages*, IEEE Computer Society Press, Kobe, Japan, pp. 4-9, 1991.

[6] Brown, M. H., and Hershberger, J., "Color and Sound in Algorithm Animation," *1991 IEEE Workshop on Visual Languages*, Kobe, Japan, pp. 10-17, 1991.

[7] Brown, M. H., and Sedgewick, R., "Techniques for Algorithm Animation," *IEEE Software*, vol. 2, no. 1, pp. 28-39, 1985.

[8] Chang, S.-K., *Visual Languages and Visual Programming*, Plenum Press, New York, NY, 1990.

[9] Cunningham, W., and Beck, K., "A Diagram for Object-Oriented Programs," *SIGPLAN Notices*, vol. 21, no. 11, pp. 361-367, 1986.

[10] Duisberg, R. A., "Animated Graphical Interfaces Using Temporal Constraints," in *Human Factors in Computing Systems: Proceedings SIGCHI'86*, ACM, Boston, MA, pp. 131-136, 1986.

[11] Duisberg, R. A., "Visual Programming of Program Visualizations: A Gestural Interface for Animating Algorithms," in *Proceedings 1987 Workshop on Visual Languages*, IEEE Computer Society, Linkoping, Sweden, pp. 55-65, 1987.

[12] Eisenstadt, M., and Brayshaw, M., "The Transparent Prolog Machine: An Execution Model and Graphical Debugger for Logic Programming," Technical Report 21a, The Open University, Milton Keynes, England, 1987.

[13] Foley, J. D., and McMath, C. F., "Dynamic Process Visualization," *IEEE Computer Graphics and Applications*, vol. 6, no. 2, pp. 16-25, 1986.

[14] Haibt, L. M., "A Program to Draw Multi-Level Flow Charts," *Proceedings of the Western Joint Computer Conference*, San Francisco, CA, pp. 131-137, 1959.

[15] Helttula, E., Hyrskykari, A., and Raiha, K.-J., "Graphical Specification of Algorithm Animations with ALADDIN," in *Proceedings of the 22nd Annual Conference on Systems Sciences*, pp. 892-901, 1988.

[16] Hyrskykari, A., and Raiha, K.-J., "Animation of Algorithms Without Programming," *Proceedings 1987 Workshop on Visual Languages*, IEEE Computer Society, Linkoping, Sweden, pp. 40-54, 1987.

[17] Knowlton, K. C., *L^6: Bell Telephone Laboratories Low-Level Linked List Language* (film), Bell Laboratories, Murray Hill, NJ, 1966.

[18] Linden, L. B., "Parallel Program Visualization Using ParVis," in *Performance Instrumentation and Visualization*, M. Simmons, R. Koskela, Eds., ACM Press, New York, NY, USA, pp. 157-188, 1990.

[19] London, R. L., and Duisberg, R. A., "Animating Programs Using Smalltalk," *IEEE Computer*, vol. 18, no. 8, pp. 61-71, 1985.

[20] Moher, T. G., "PROVIDE: A Process Visualization and Debugging Environment," *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 849-857, 1988.

[21] Moriconi, M., and Hare, D. F., "Visualizing Program Designs Through PegaSys," *IEEE Computer*, vol. 18, no. 8, pp. 72-86, 1985.

[22] Myers, B. A., "Incense: A System for Displaying Data Structures," *ACM Computer Graphics (Proceedings SIGGRAPH'83)*, vol. 17, no. 3, pp. 115-125, 1983.

[23] Myers, B. A., "Taxonomies of visual programming and program visualization," *Journal of Visual Languages and Computing*, vol. 1, no. 1, pp. 97-123, 1990.

[24] Radack, G. M., and Desai, T., "Akrti — A System for Drawing Data Structures," CES-88-06, Case Western Reserve University, 1988.

[25] Reiss, S. P., "Working in the Garden Environment for Conceptual Programming," *IEEE Software*, vol. 6, no. 6, pp. 16-27, 1987.

[26] Roman, G.-C., Cox, K. C., Wilcox, C. D., and Plun, J. Y., "Pavane: a System for Declarative Visualization of Concurrent Computations," *Journal of Visual Languages and Computing*, vol. 3, 1992.

[27] Roman, G.-C., and Cox, K., "A Declarative Approach to Visualizing Concurrent Computations," *Computer*, vol. 22, no. 10, pp. 25-36, 1989.

[28] Roman, G.-C., Cox, K. C., and Boemker, T. J., *Diffusing Computations* (VHS video tape), Computer Visualization Laboratory, Washington University, St. Louis, Missouri, 1990.

[29] Shu, N. C., *Visual Programming*, Van Nostrand Reinhold Company, New York, NY, 1988.

[30] Stasko, J., "Simplifying Algorithm Animation with TANGO," *IEEE Workshop on Visual Languages*, IEEE, Skokie, IL, USA, pp. 1-6, 1990.

[31] Stasko, J., "Using Direct Manipulation to Build Algorithm Animations by Demonstration," *CHI'91 Human Factors in Computing Systems*, New Orleans, pp. 307-314, 1991.

[32] Tufte, E. R., *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, CT, 1983.