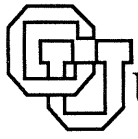
Visualization-Based Visual Programming

Wayne Citrin

CU-CS-535-91 July 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Visualization-Based Visual Programming

Wayne Citrin

CU-CS-535-91

July 1991

Wayne Citrin
Dept. of Electrical and Computer Engineering
Campus Box 425
University of Colorado
Boulder, CO 80309-0425

citrin@soglio.colorado.edu

tel: 1-303-492-1688
fax: 1-303-492-2758

Visualization-Based Visual Programming

Wayne Citrin

Department of Electrical and Computer Engineering
Campus Box 425
University of Colorado, Boulder 80309-0425
USA

citrin@soglio.colorado.edu

ABSTRACT

A new class of visual programming languages is proposed which employs paradigms typical of program visualization as the programming language itself. Such paradigms, called visualization-based visual programming, should improve programmer productivity by allowing programs to be written in a form corresponding more closely to the programmer's mental models of the problem and its solution. Two examples of such a language are given: one for network protocol specifications, and one for data structure manipulation.

1. Introduction

Over the last 5-10 years, the advent of workstations with bit-mapped graphic displays and pointing devices has created the possibility of new types of interaction with the computer. Previously, programming was necessarily a one-dimensional activity, where a program was a string of text supplied by the programmer through a keyboard. (Admittedly, such programs have line breaks and are displayed as multiple lines on a terminal screen, but this does not conceal the fact that they are character strings.) Likewise, debugging output describing the progress of a running program was a sequence of lines of text describing program activity, changes in data structures and program state, and so forth. Bit-mapped displays and pointing devices free the programmer from the necessity of describing programs through text, and allow the programmer to describe programs and data structures multi-dimensionally, through pictures drawn on a screen and entered through the use of a pointing device such as a mouse, joystick, track ball, or light pen. In the other direction, the progress of running programs can now be displayed to the user through the use of pictures when such pictures may be used to lend greater insight into the activities of the program.

Myers[18] has classified visual programming activities into two large subclasses: visual programming proper, and program visualization. Visual programming is the communication of programs or data to the computer through the use of pictures. It generally consists of the manipulation of icons[5] representing program structures which often correspond to textual program structures. Such program structures may be far removed from the mental structures through which the programmer visualizes the algorithm[2]. On the other hand, program visualization systems often abstract out low-level program structures and display

information on program activities in a form closer to the user's or programmer's mental models, thus lending greater insight into the activities of the program, and allowing the programmer to determine whether the program is functioning as intended.

The gap between paradigms used in visual programming and program visualization is a fundamental problem in the field of visual programming, limiting its usefulness. This paper proposes a new class of visual programming languages, known as visualization-based visual programming, in which paradigms useful for visualization, that is, corresponding to high-level abstract visualizations of program activity, may be used for actual programming. We will discuss the advantages of such a scheme, identify the characteristics of those visual paradigms that are amenable to such use, and discuss two examples of visualization-based visual programming languages.

2. Previous work

In the strictest sense[18], visual programming is the use of pictures to communicate programs and data to a computer. Most visual programming systems are *iconic*[4, 5, 14], although some are *forms-based*[21, 23]. In an iconic system, icons representing data or program functions are juxtaposed in various ways in order to specify program activities. In certain systems[5, 22], icons representing program functions are superimposed on, or placed next to, icons representing data in order to denote the function applied to the data. Icons may be composed to create new icons, a notion analogous to procedure abstraction. In other systems, icons representing program activities are connected by lines representing flow of control[14] or flow of data[4]. Such visual programs will represent flow charts or data-flow graphs, respectively. While use of visual languages like these may yield insight into program structure, they are still direct analogues to underlying textual programs, and the act of programming in such systems is no less difficult than that of programming in the corresponding textual paradigms.

One response to the above was the proposal of conceptual programming[20]. In such systems, the visual entities do not correspond to any common textual program structures. The most famous conceptual system is the Garden system developed by Reiss[20], and the example most commonly provided in the literature on Garden is the finite state machine presented as a transition graph. While such a system is not inherently textual, it is still inherently procedural, and of a similar order to programming in a procedural textual language, although the task is simplified by the fact that the programmer no longer needs to be concerned with textual syntax, but rather can concentrate on the state changes and transitions in the system being developed. (It is conceivable that visualization-based languages of the type described in this paper could be implemented in Garden, but languages of this type have not been found in the Garden literature.)

Another visual programming paradigm attempts to avoid many of the difficulties of programming by allowing the user to directly manipulate objects representing the program input and output. The system attempts to generalize from these examples and produce a complete program. Such *programming-by-example* systems[12, 15] have the drawback that the systems cannot generalize sufficiently and completely to produce a useful or even correct program without extensive intervention by the user, and it appears that they cannot be scaled up to be useful in solving real-world problems, except in certain specific domains, such as user interface design[17, 19].

Program visualization systems, on the other hand, attempt to model program activity in a form closely corresponding to the programmer's or user's mental image of the program. For example, one system[11] attempts to model a running Prolog program as a traversal of an AND/OR tree presented graphically on a screen. Another system[9] displays traces of executing communication protocols as message flow diagrams. A third system[16] presents performance information for parallel systems as a set of parallel time lines each colored or shaded to represent a process's state at a given point in time. Such paradigms correspond closely to mental images of the activities in question (see section 3) and can convey a great deal of insight and information.

Certain *direct manipulation* systems come closest to employing useful visualization paradigms to instruct the computer. A classic example is a system which displays entity-relation schema describing databases, and models queries on the database as transformations on those schema[10]. Such systems are not general enough to be useful for programming, however, since they do not allow operations to be composed or otherwise combined, in order to provide more complex abstractions.

3. Visualization-based visual programming

In *visualization-based visual programming*, visual representations corresponding to mental models are used for programming. There are a number of advantages to this. Ackerman and Stelovsky[2] describe programming as the act of mapping mental models into programming language constructs and the main problem of programming as the gap between the two. They suggest that the problem should be alleviated by providing tools to assist in the transition. We suggest that the answer lies in programming language constructs that correspond closely to mental models, thereby reducing or even eliminating the conceptual gap.

This begs the question of how such models can be identified. Ackerman and Stelovsky suggest that it can only be done through interviews with programmers and by careful examination of the code they produce. We propose that such models can be found by examining documentation, manuals, and textbooks. When a communications protocol designer wishes to describe a protocol he or she has designed, he or she often does not use a programming language, or even a formal specification language, but rather illustrates the operation of the protocol through through use of a message flow diagram[1]. Likewise, a basic Prolog textbook[8] explains the operation of Prolog programs to beginners through the use of trees. General programming language and compiler textbooks[13] illustrate complex data structures through the use of labeled boxes and arrows connecting them. It is no coincidence that the visualization systems described in the previous chapter use these models, since they correspond closely to the programmers' mental models.

Assigning executable semantics to the diagrams corresponding to mental visualizations (that is, making them programming languages) would solve a basic problem of the programming process, whether the programs are written using visual or textual paradigms. A programmer must currently transform his or her mental models into programming language constructs. When debugging the program, if a visualization system is being used, the programmer must match the visualization information and the program, which are likely to be in different paradigms. If no visualization system is being used (for example, if a textual debugger is being used), the programmer must map *both* the program *and* the debugging information into his or her mental models. Each mapping, or *paradigm shift* takes time and detracts from programmer

productivity. It is also likely to introduce mistakes, which will show up either in erroneous code, or in bugs that are not identified during debugging. The purpose of visualization-based visual programming is to reduce, or even eliminate, the paradigm shift by merging the programming and visualization models.

This class of paradigms has a number of other advantages. Because the program resembles the programmer's (and possibly the user's) mental models, the program can serve to a large extent as documentation. (We do not claim, however, that such programs are self-documenting.) The paradigm might also simplify the act of fixing bugs. Since programs and visualization are in the same form, if a running program produces incorrect or unanticipated behavior in the visualization, the programmer can edit the visualization to exhibit the correct behavior and submit the *visualization* to the compiler as program source. The compiler can then identify where the newly submitted program differs from the original program and make the appropriate corrections.

4. Visualization-based visual programming paradigms

It is important to identify visual paradigms which are suitable for visualization-based visual programming. One method is by searching the literature for illustrations, as mentioned above. The search can be narrowed and pointed in certain directions. The scope of a paradigm may have an effect on its usefulness: namely, the paradigm may be (textual) language-based, application domain-based, or program-based. For the most part, visual paradigms that are mapped closely to textual languages (for example, flowcharts) are not appropriate since they do not map well to mental models and lead to a paradigm gap in the programming process. A few language-based paradigms, where the paradigm does correspond to a mental model, may be suitable. One such model is the aforementioned AND/OR trees based on Prolog. Another is a model related to object-oriented programming presenting class hierarchies as a directed graph and showing a program as a network or objects with messages passing back and forth between them.

Paradigms based on application domains seem to be a fertile area for investigation, since people who work in a given domain often develop a common set of visual paradigms to explain their work to each other. One such domain is communications protocols, with its message flow diagrams, network graphs, and protocol towers. Another is the E-R graphs of entity-relationship databases. Tables are an appropriate visual paradigm for relational databases. Other domains containing useful visual paradigms are VLSI design, hardware configuration, and robotics.

Program-based paradigms, on the other hand, reduce the usefulness of visual programming when each program has its own visual paradigm. In the network domain example, to would be much more useful if a protocol compiler, validator, tester, and simulator all accepted the same visual input. Having them each accept different forms of input representing the same thing would lead to another paradigm gap.

Visualization-based visual programming paradigms can be classified in another way: they may be output-oriented or state-oriented. A message flow diagram is an output-oriented paradigm. It displays the visible behavior of the protocol as a trace of messages. (As will be seen in the next section, message flow diagrams also incorporate state information; few practical paradigms are purely output- or state-oriented.)

The AND/OR tree visual paradigm for Prolog is a state-oriented paradigm. The tree describes the state of the computation, and the language paradigm describes the transition between one state and another (or perhaps between the relevant part of one state and the corresponding part of the resulting state). Unlike

the output-oriented paradigms, which can express "motion" or progress in a single picture, state-oriented paradigms like AND/OR trees show progress by a series of "snapshots;" a transition is a pair of pictures (and perhaps some additional enabling information) describing the "before" and "after" situation of a portion of the state.

5. An example - Message-flow diagrams

We present a simple example of a visualization-based approach using message flow diagrams. In figure 1, there are two network nodes: N1 and N2. N1 contains a state variable X. N2 contains no state variables (or at least no relevant ones).

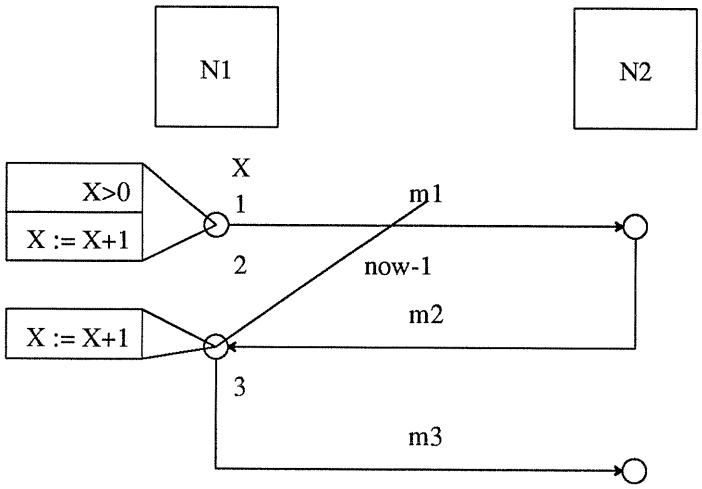


Figure 1. A program based on a message-flow diagram

At its most superficial level, the diagram says that, in a certain conversation, a node of type N1 will send a message m1 to a node of type N2. When N2 receives it, it replies with message m2. On receipt of m2, N1 replies with m3.

The diagram has been augmented with state and dependency information. The initial message m1 is only sent when the state variable X is greater than zero. After the message is sent, X is incremented. When N2 receives m1, it always responds with m2. If N1 receives m2 and it previously sent message m1 during the previous clock cycle (denoted by the dependency edge labeled "now-1", N1 sends message m3 and increments X.

We see two possible methods of interpreting this diagram. The first is a rule-based interpretation. The firing rules described above are stored in a rule database, and a network entity fires a rule associated with its type in the database whenever the enabling conditions are met (which means that the rules might be fired in an order different from that in the defining diagram). The second interpretation is a script-based interpretation. In this interpretation, the diagram is a script that is executed as long as the enabling conditions are met. If a condition is not met, the diagram is abandoned. Several consistent diagrams may be active simultaneously, provided they are equivalent up to that point according to some set of criteria. This scheme would lend itself well to a translation into a process algebra-based language like LOTOS[3]. We propose implementing both interpretations and determining through tests which one most closely

corresponds to programmers' mental models.

For the purposes of brevity, the above example was necessarily simple. Diagrams may be parameterized, and messages may contain variables. Diagrams may be composed, so that when execution runs off the bottom of one diagram, it commences at the beginning of another. Multiple conversations (or diagrams) may be active simultaneously. Recursion and iteration may be specified. The diagrams also lend themselves to multi-level abstraction: the transmission and receipt of the message m1 at this level might correspond to an extensive exchange of messages at a lower level. Similarly, the receipt of m1 and the resulting transmission of m2 might correspond on a lower level to a conversation between entities which are components of N2.

A more detailed description of a language based on message-flow diagrams is given in[6].

An environment for executing or simulating programs written in this language would, of course, present the visualization of the running program in a form identical to that of the program itself: a message flow diagram.

6. Another example - data structure manipulation

Another simple example of a visualization-based language involves the almost ubiquitous visual model of data structures as boxes and arrows. The following example is very sketchy, but should suggest the flavor of such a language.

In the first example (figure 2), the act of pushing a value onto a stack is specified. There are two diagrams, indicating the ''before'' and ''after'' states. In the ''before'' state, a value is about to be pushed onto a stack. The value is represented simply by the symbol X. The stack is represented by a box (denoting a memory location) next to another box containing an ellipsis (indicating that there are zero or more instances of the memory location denoted by the first box). This suggests an array.

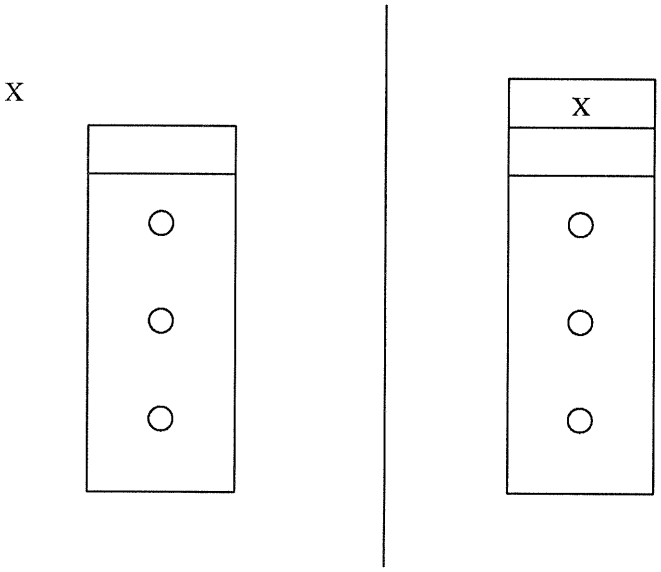


Figure 2. Stack push: "before" and "after"

In the ''after'' picture, X is placed in a new box above a copy of the old array. Being placed above suggests that the stack grows upwards. There is a question as to how to indicate that the array in the ''before'' picture is the same as the array in the ''after'' picture. One possibility is to draw a line between the two. Another is to simply to point to both of them (assuming that the environment allows gestures) or to assign them a common name.

The second example (figure 3) shows a step in an insertion sort algorithm in a linked list. When the value to be inserted falls between the first and second elements of the list, we insert it. Ellipses indicate a possible continuation of the list.

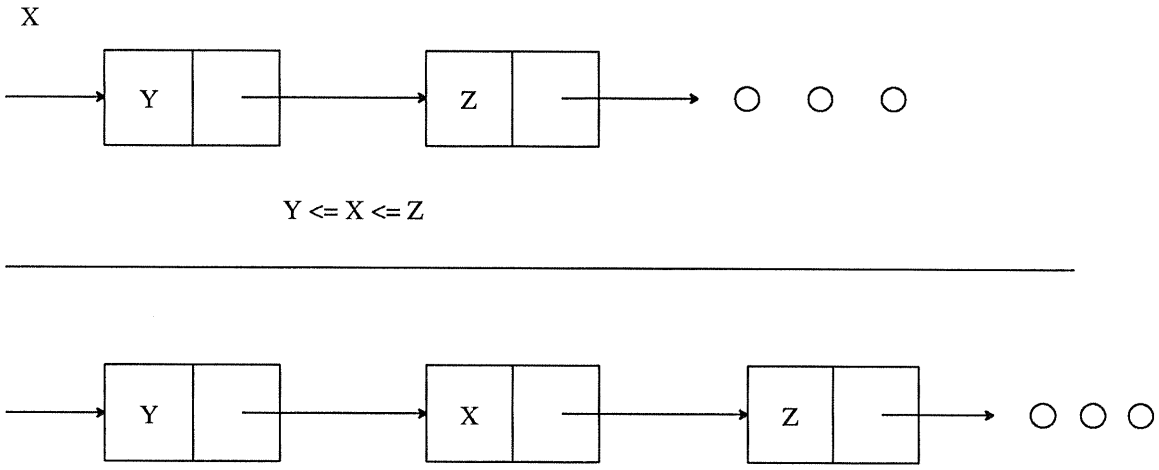


Figure 3. List insertion: "before" and "after"

The purpose of this section is simply to give an impression of what such a language might be like. There are clearly further problems to be solved, particularly in the specification of recursion and iteration.

7. Current and future work

Over the past two years, a system, known as Cara[9], has been developed that used message flow diagrams as a form of specification. In that project, we did not attempt to assign an executable semantics to message flow diagrams. Instead, for each action drawn on the screen, the system used a set of heuristics to construct a textual rule describing the action in a language called Carla[7]. The rule was then displayed to the user, who was free to edit it as he or she pleased, before depositing it in the rule database. The system could also simulate the rules, displaying the results in the form of a message flow diagram. The Cara system was intended as a specification aid, rather than a programming environment.

Currently we are attempting to demonstrate the feasibility of the visualization-based visual programming approach by creating a graphical language based on message flow diagrams, with a well-defined syntax and semantics. (In the Cara system, diagrams had neither well-defined syntax nor semantics.) Following this, we will implement a programming environment using message flow diagrams both as programming language and visualization paradigm, and attempt to implement complex protocols using this method.

In parallel to this, we plan to identify and develop other paradigms of this type. A language for data structure manipulation based on the box-and-arrow model seems a likely candidate. We also plan to investigate methods of specifying such visual languages formally and generating programming environments

automatically from such language specifications.

References

1. *Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2,* Document SC30-3269-3, IBM Corporation, December 1985.
2. D. Ackerman and J. Stelovsky, "The Role of Mental Models in Programming: From Experiments to Requirements for an Interactive System," *Visualization in Programming*, pp. 1-23, Springer-Verlag, Berlin, 1987. Selected contributions, 5th Interdisciplinary Workshop in Informatics and Psychology, Schaerding, Austria, May 1986
3. T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS," *Computer Networks and ISDN Systems*, vol. 14, pp. 25-59, 1987.
4. A. Borning, "Defining Constraints Graphically," *Proc. CHI 86 - Human Factors in Computing Systems*, pp. 137-143, April 1986.
5. S.-K. Chang, "Principles of Visual Languages," *Principles of Visual Languages*, pp. 1-59, Prentice-Hall International, Englewood Cliffs, NJ, 1990.
6. W. Citrin, "Design Considerations for a Visual Language for Communications Architecture Specifications," *Proceedings 1991 IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991. to appear
7. W. Citrin and A. Cockburn, "An Executable Specification Language for History-Sensitive Systems," *IBM Zurich Research Laboratory Research Report*, no. RZ 2162, July 1991.
8. W. Clocksin and C. Mellish, *Programming in Prolog,* Springer-Verlag, Berlin, 1981.
9. A.A.R. Cockburn, W. Citrin, R.F.Hauser, and J. von Kaenel, "An Environment for Interactive Design of Communications Architectures," *Proc. 10th Intl. Symposium on Protocol Specification, Testing, and Verification*, Ottawa, June 1990.
10. B. Czejdo, R. Elmasri, M. Rusinkiewicz, and D. Embley, "A Graphical Data Manipulation Language for an Extended Entity-Relationship Model," *IEEE Computer*, pp. 26-36, March 1990.
11. M. Eisenstadt and M. Brayshaw, "The Transparent Prolog Machine (TPM): An Execution Model and Graphic Debugger for Logic Programming," *J. Logic Programming*, vol. 5, pp. 277-342, 1988.
12. W. Finzer and L. Gould, "Programming by Rehearsal," *Byte*, vol. 9, no. 6, pp. 187-210, June 1984.
13. C. Ghezzi and M. Jazayeri, *Programming Language Concepts (2nd Ed.),* John Wiley, New York, 1987.
14. E. P. Glinert and S. L. Tanimoto, "Pict: An Interactive Graphical Programming Environment," *Computer*, pp. 7-25, November 1984.
15. D. Halbert, *Programming by Example,* Computer Science Division, University of California, Berkeley, CA, 1984. PhD Thesis
16. T. Lehr, Z. Segall, D. Vrsalovic, E. Caplan, A. Chung, and C. Fineman, "Visualizing Performance Debugging," *IEEE Computer*, pp. 38-51, October 1989.

17. D.L. Maulsby, I.H. Witten, and K.A. Kittlitz, "Metamouse: Specifying Graphical Procedures by Example," *Proceedings SIGGRAPH '89*, Boston, July 1989.

18. B.A. Myers, "Taxonomies of Visual Programming and Program Visualization," *Journal of Visual Languages and Computing*, vol. 1, pp. 77-95, 1990.

19. B. A. Myers, "Creating Interaction Techniques by Demonstration," *IEEE Computer Graphics and Applications*, pp. 51-60, September 1987.

20. S. P. Reiss, "Garden Tools: Support for Graphical Programming," *Advanced Programming Environments*, pp. 59-72, Springer-Verlag, Berlin, 1986.

21. N. C. Shu, *Visual Programming*, Van Nostrand Reinhold, New York, 1988.

22. K. Tsuda, A. Yoshitaka, M. Hirakawa, M. Tanaka, and T. Ichikawa, "IconicBrowser: An Iconic Retrieval System for Object-Oriented Databases," *Journal of Visual Languages and Computing*, vol. 1, pp. 59-76, 1990.

23. M. M. Zloof, "Query-by-Example: A Data Base Language," *IBM Systems Journal*, vol. 16, no. 4, 1977.