USING PROGRAM VISUALIZATION TO ENHANCE MAINTAINABILITY AND PROMOTE REUSE

Chuck Ames
Jet Propulsion Laboratory
Pasadena, Ca 91109

Lizz Howard
Department of Computer Engineering
University of Cincinnati
Cincinnati, Oh

James D. Kiper, PhD
Department of Systems Analysis
Miami University
Oxford, Oh 45056

Abstract

Our intuition is that comprehension of visual representations is often quicker than of equivalent text. In the work described in this paper, we explore the application of this intuition to programming languages. The goal of this project is to create visual representation of segments of computer programs that improve the maintenance and reusability of this code.

We describe a software tool, a program browser, that provides a visual representation of the function call graph of any C program. This tool is the first in a series of tools that aids program comprehension, making reuse of existing programs more likely.

The development of the program browser is itself an example of code reuse as described in the section on implementation.

Although our intuition is that some visual display of programs aid understandability, we are looking for experimental verification. Several human factors experiments in this area have contradicted this intuition. We describe reasons for these contradictions, and potential solutions to overcome them.

1. Introduction

This paper describes a project which is exploring the use of program visualization as a means of enhancing program comprehensibility. Our goal is to create a set of visual mechanisms that will ultimately help reduce maintenance cost, make code reviews more effective, and promote re-use. Our general approach has been to create a visual representation for C programs. More specifically, we are developing a graphical syntax that is semantically equivalent to the C language, and building a tool to display C language code in the new graphical syntax.

Program comprehension is perhaps the most important factor affecting the efficiency of maintenance programmers [2,3]. The entire software engineering philosophy is built on the premise that high level language code is created for human consumption rather than driven by machine requirements. The first step in repairing or modifying existing code is to understand what that code does.

Comprehension is also a necessary pre-requisite to re-using code. The object-oriented paradigm achieves its reuse potential by designing that potential into a system through intelligent use of encapsulation and inheritance hierarchies. Legacy code that was not designed for reuse must be re-engineered if its reuse potential is to be exploited. Again, the first step in this re-engineering process is understanding the meaning and purpose of existing code.

Thus, mechanisms for improving program comprehensibility can reduce maintenance cost and maximize return on investments in legacy code by promoting reuse.

2. Enabling Tool Reuse Implementation Example

The first phase of this project has focused primarily on the development of a high level graphical C program browser. The browser shows gross program structure in the form of a call graph generated directly from the program code. Each node in the call graph represents a user defined function, and each arc represents a "calls" relationship. (See figure 1.) Clicking on a function node brings up the corresponding C code in a window.

The program browser is based on a modified copy of the GNU C compiler. The modified compiler outputs a text representation of the call graph which is then displayed by a rendering tool. Direct recursion, indirect recursion, and functions called through pointers are supported. Control constructs (e.g. 'if' and 'while') will be added to enhance the call graph in the future. Ultimately, this approach will be used to create an abstract representation of the implementation of each function that can then be displayed graphically as well, thus providing a completely visual representation of C programs.

The program browser development is itself an interesting case study in re-using tools for rapid prototyping. The initial prototype used the Pic language [6] to lay out the call graphs. Pic code was then converted to Postscript using the GNU groff package, and displayed using the GNU Postscript interpreter, Ghostscript. The current prototype is an interactive X-Window application built in Tcl/Tk to take advantage of the interactive point-and-click capabilities offered by Tk [1].

Currently the program browser obtains line numbers of textual program code from the compiler and uses this information to display a function in a window when the user clicks on that function in the call graph. In subsequent versions, this text representation will be replaced by a graphical representation. This graphical representation will serve as the basis for a study to determine if program visualization is effective for increasing program comprehensibility. If this hypothesis proves correct, program visualization may be useful for reducing maintenance cost and promoting software re-use.

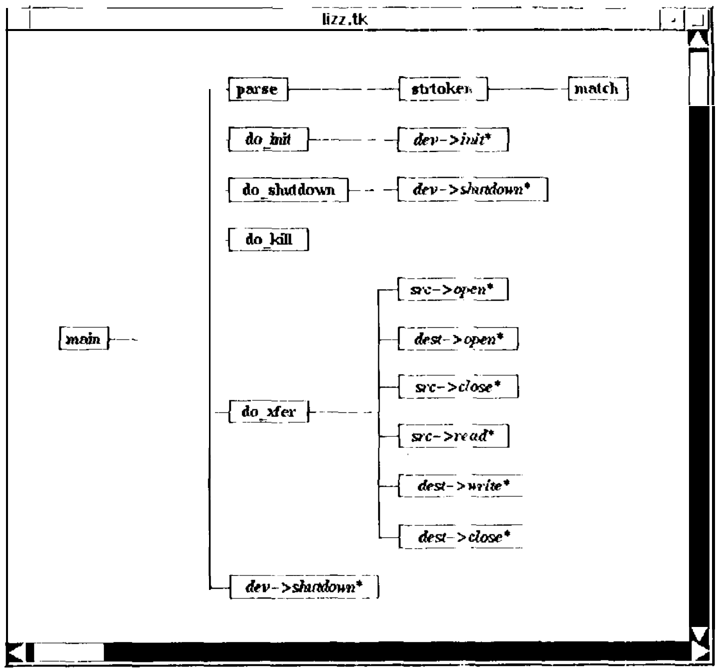


Figure 1: Program Browser Output.

3. Enabling Tool Reuse through Visual Programming

In this section we address the more general issue of visual programming as an aid to code reuse. We first describe the importance of program comprehension and the relationship of the functionality of the program browser to comprehension. Next we present some of the most important research efforts in the area of visual programming that are related to this project. Finally, we describe the results of some human factors experiments in this area, and describe how these results are affecting our language design.

3.1 Program Browser and Program Comprehension

Reading and understanding a program is prerequisite to any review or maintenance task. It is generally agreed that errors found in reviews are much less costly to fix than the same errors not found until after delivery. It is also generally agreed that the majority of programming activity can be classified as maintenance programming. Therefore, improving the efficiency with which programmers read and understand code could have a significant impact on development and maintenance cost and schedule.

We believe that representing programs visually will make them accessible to a wider audience, specifically technics/ *non-programmers*. Using visual code printouts to augment textual listing allows domain experts who are not necessarily programmers to contribute to program reviews, This additional perspective could help reveal errors that would other wise go unnoticed.

Secondly, we expect the program browser to be helpful to maintenance programmers. As shown in figure 1, the browser provides a simple view of gross program structure in terms of the way various functions are related. We already have anecdotal evidence to demonstrate that this alone is a tremendous time saver. Users can click on a call-graph node to display the code of the function associated with that node in another window. Future versions of the browser will allow the user to view the implementation of a function either as textual or visual code.

3.2 Visual Programming Background

Singh and Chignell[14] provide a taxonomy of the area that divides visual computing into three main areas: *programming computers, end-user interaction with computers,* and *visualization. (See* Figure 2). The first branch of the taxonomy, the area of programming computers (visual programming and program visualization), is the main concern of a program designer. End user are most concerned with the user interface, the second branch of the taxonomy. Users who must interpret and process large amounts of data, especially scientists and data analysts, are most interested in the last branch of the taxonomy, scientific visualization. The main focus of the work described in this paper is in the first branch of the taxonomy, specifically visual programming.

Singh and Chignell have divided the first branch, *programming computers,* into two key areas: *visual programming* and *visualization. (See* Figure 3). The generally accepted definition of visualization is the use of various techniques to aid in the understanding and debugging of computer programs [1]813,14, Visual programming, on the other hand, "refers to any system that allows the user to specify a program in a two (or more) dimensional fashion. Conventional textual languages are not considered two dimensional since the compiler or interpreter processes it as a long, one-dimensional stream. " 8

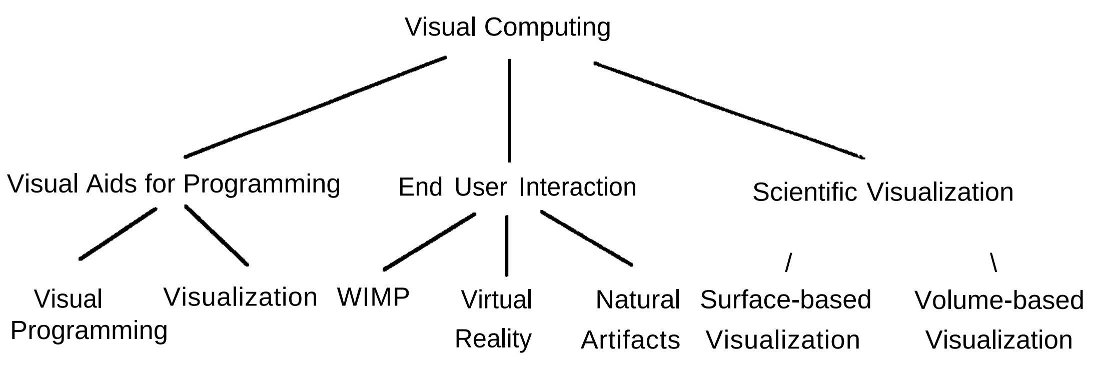


Figure 2. Singh and Chignell's[14] Classification of visual computing

Singh and Chignell[14] have further divided visualization into three main branches: *program visualization, algorithm visualization,* and *data visualization.* In program visualization, graphics are used to illustrate some aspect of the program after it is written and can be either *static* or *dynamic program visualization.* Static program visualization techniques include flow charting and *pretty-printing* (insertion of blanks and blank lines, indentation, and comments to enhance the readability of a program). Execution of the program is illustrated either by animation or by highlighting the program code when dynamic program visualization is implemented.

Algorithm visualization systems produce animations of algorithms in order to show the "program fundamental operations that embody both transformations and accesses to data and flow-of -control." [14].

Data visualization can also be subdivided into either *static* or *dynamic data visualization.* Static data visualization generates static pictorial displays for data structures. This method makes "debugging easier by presenting data structures to programmers in the way that they would draw them by hand on paper. "[8]. As the name implies, dynamic data visualization graphically displays the state of variables, arrays, lists, trees, and other data structures as the program is executed.

In essence, *visualization* provides a means to better understand how a program works after it has been coded. This is in direct contrast with *visual programming,* where a program is actually designed by manipulating graphical representations (icons) or by a combination of icons and textual information,

Singh and Chignell[14] divide visual programming into two key branches: *graphical interaction systems* and *visual language systems. (See* Figure 3). This division is based upon how the graphics are used to build the program. Systems where the user guides or instructs the system in order to create the program are classified as graphical interaction systems. Visual language systems consist of systems in which icons, symbols, charts, or forms are used to specify the program.

In graphical interaction systems, the sequence of user actions is of vital importance since the system "learns" from the user input. This category is more commonly called *programming by example.*

In the majority of systems, a user is required to specify everything about the program and the system is able to remember the examples for later use. This type of system could be described as *"Do What / Did" 8*. Conversely, some systems attempt to infer the general program structure after the user has provided a number of examples which work through the algorithm. These systems could be characterized by "Do *What/ Mean"* [8] and are often referred to as automatic programming, which has generally been an area of Artificial Intelligence research.

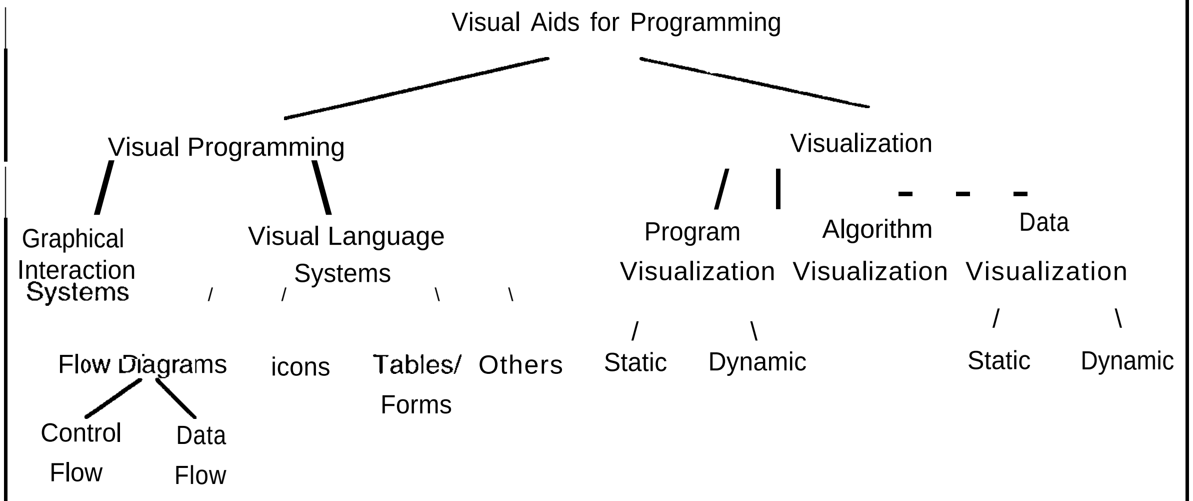


Figure 3. Singh and Chignell's[8] Taxonomy of visual programming systems

The second branch of visual programming is *visual language systems.* Within this classification are systems using *icons, symbols, charts,* and *forms* to specify the program. The spatial arrangement of the symbols specifies the program. This differentiates visual languages from graphical interaction systems (programming by example), since, in graphical interaction systems, the user interaction with the system is important, and in visual languages, the arrangement of symbols on the screen is important.

Visual languages are composed of a set of graphical symbols which are constructed into "visual sentences with a given syntax and semantics. "[4]. Visual sentences must then be spatially parsed to determine the underlying syntactic structure. A semantic analysis must then be performed to determine the meaning of the visual sentences (spatial interpretation). The syntactic and semantic analyses of a visual language differs little from a traditional language approach. Both types of languages must be analyzed to determine syntax and meaning, the significant difference being that visual languages employ graphical symbols rather than textual expressions of traditional languages, In Figure 3, Singh and Chignell[14] suggest a division of visual languages into three main categories based upon the graphical abstraction used for creating the program: *flow diagrams*, *icons,* and *forms/tab/es.*

The category, *flow diagrams,* includes visual languages which provide various types charts, graphs, and diagrams to construct programs. Flow diagrams are primarily composed of *control flow* or *data flow diagrams.*

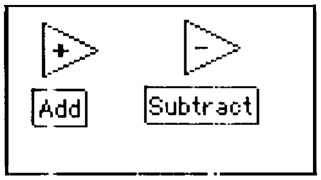
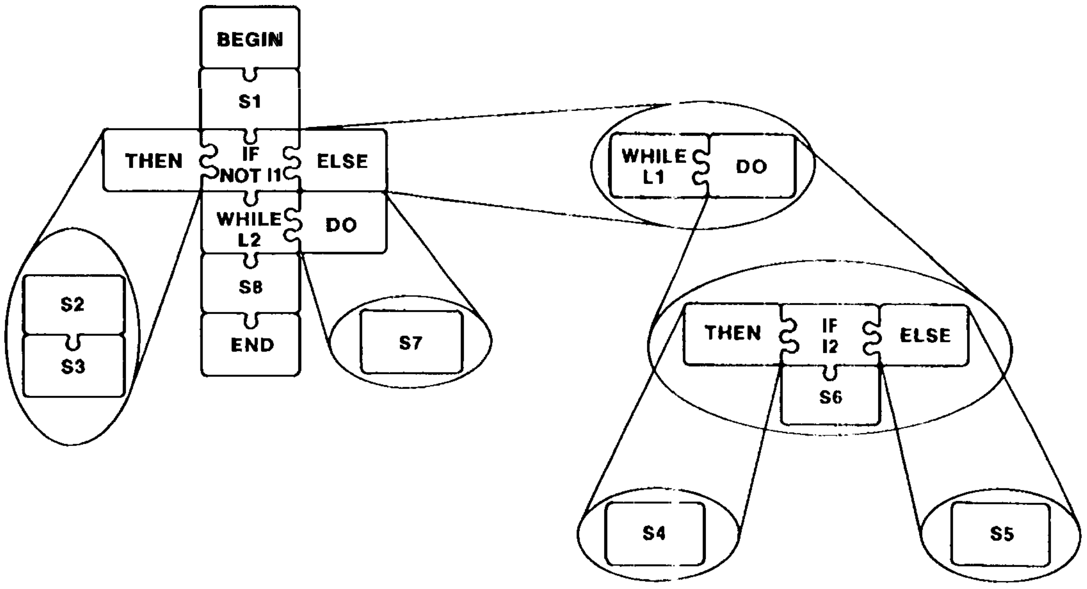


Figure 4. LabVIEW icons for add and subtract functions



```
BEGIN
          S1;
          IF NOT II THEN
                    BEGIN
                              S2;
                              S3;
                    END
          ELSE
                    WHILE L1 DO
                              BEGIN
                                        IF 12 THEN S4 ELSE S5; S6;
                              END;
          WHILE L2 DO S7;
          S8;
END;
```

Figure 5. Proc-BLOX Representation and Pascal Program Equivalent

The most common example of control flow diagrams (and probably the earliest visual representation for a program) is the flow chart. Typically, the flow chart was used for documentation purposes, but visual languages employing flow charts create programs automatically. Another type of control flow diagrams used in some" visual languages are Nassi-Shneiderman diagrams[9]. Data flow diagrams depict the flow of data from one operation or object to another and the visual language constructs the program from the flow of data.

The second category of visual programming languages, *icons*, consists of visual languages using graphical symbols or icons and their interconnections to form visual sentences. As was noted earlier, spatial parsing and interpretation is used to provide syntactic and semantic analyses, respectively. There is no accepted standard for the definition of an icon. The main criterion for designing an icon is that it clearly represents the abstraction. For example, in LabVIEW[10], the traditional symbol for an operational amplifier is used to represent the functions add and subtract. (See Figure 4). Another use of icons is in the language Proc-BLOX[5]. Figure 5 illustrates a Proc-BLOX implementation of a traditional Pascal program, where the Proc-BLOX symbols resemble a three dimensional jigsaw puzzle.

The final category of visual languages are those languages which employ *tables* and *forms.* The user constructs the program by using tables or filling in forms. Common uses of this category include developing queries on relational databases through the use of tables and the development of office-information systems using forms.

3.3 Results of Experiments

Over the past few years several human factors experiments have been conducted to determine whether visual representations of program segments increase the comprehensibility of that program by users.[7, 12]

The conclusions of these experiments seem to run counter to our intuition that visual representations should aid comprehension. However, a deeper look at these experiments reveals that their purpose was to disprove the superlative statement that all visual representations are better than textual formulations. We have taken the hypothesis that there probably exists some visual representation that is generally better for comprehension for a specific task than text.

In task one, subjects were presented both "forward" and "backward" questions. For forward questions, subjects were given a set of input conditions along with a decision tree, and were asked for the result of the tree. Backward questions gave subjects a conclusion of the given decision tree and asked for the truth or falsehood of input conditions that would lead to this conclusion.

Table 1: Task 1 Results

mode	form	average time	stdev
text 1	forward	74.09	47.72
text 2	forward	74.55	29.32
gates	forward	285.44	358.83
boxes	forward	142.48	226.91
text 1	backward	76.06	44.78
text 2	backward	44.97	16.37
gates	backward	217.53	114.78
boxes	backward	171.27	116.06

Table 1 gives results of this experiment. It is clear that subjects performance on both graphical forms was significantly worse than on textual form. This conclusion is the same as that of prior experiments.

A closer look at the form of textual and graphical representations reveals that the textual forms were much better adapted to the questions than were the graphical forms. For example, input conditions corresponding to a given output, i.e. a backward type question, can be simply read off from the second textual form. To answer the same question using the gates graphical form requires a user to follow lines through a complex maze of crisscrossing "wires." The boxes graphical form is somewhat easier, but still requires a user to examine several nested box combinations before arriving at the appropriate conclusion. We believe that subjects performance on graphical forms would be "much better if graphical representations were used that correspond more closely to the question asked. We plan further experiments in this area to explore this hypothesis.

```
if high :
   if long : laugh
   not long :
        if thick :    whistle
        not thick : cry
       end thick
   end long
not high :
   if wide :    cry
   not wide : shout

end high
```

Figure 6: Text Type 1 Example

We have verified the results of these experiments by a replication using our own subjects. The experiment involves two tasks: the first requires each subject to determine the result of a decision tree represented graphically or textually; the second requires each subject to compare a graphical and a textual form of a decision tree to determine is these are the same or different. For each task, two different textual format, and two different visual representations were used. Examples of each of these forms are given in figures 6 though 9.

```
grill :  if red & orange
fry :     if ¬ red & ¬ blue
boil :    if ( --, red & blue ) |
          ( red & ¬ orange ¬ green )
roast : if red & ¬ orange & green
```

Figure 7: Text Type 2 Example

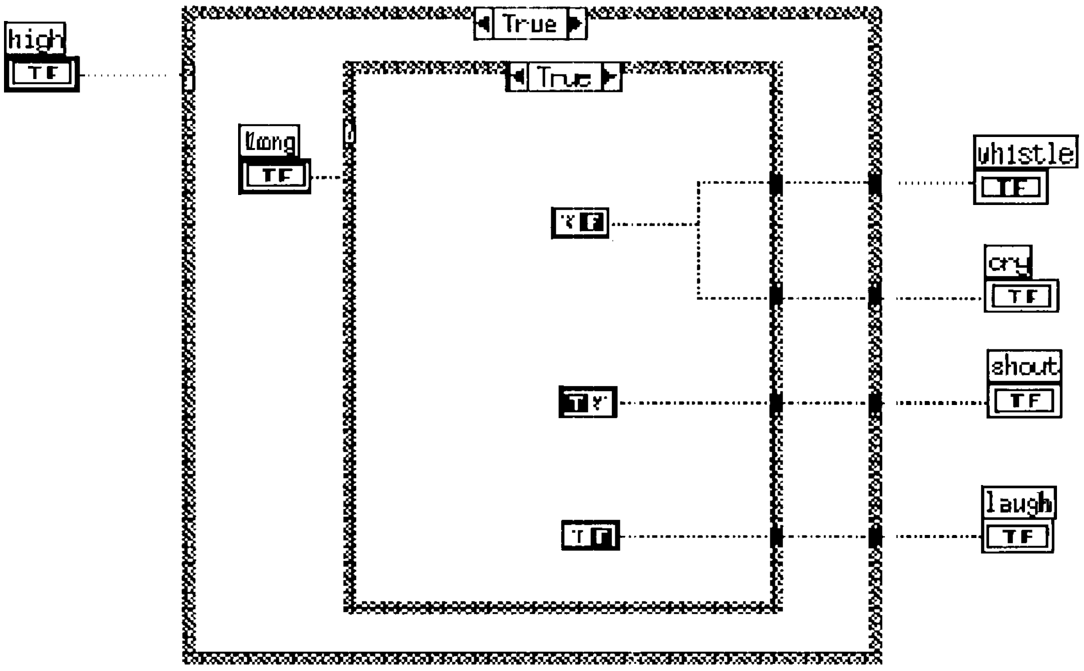


Figure 8: Boxes Graphical Form Example

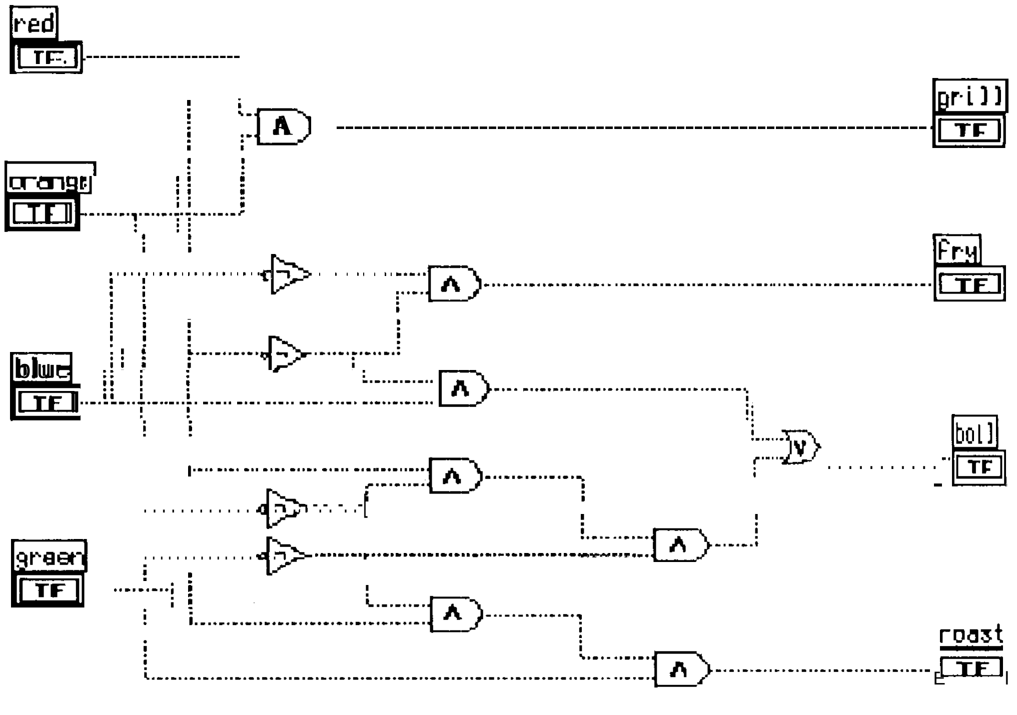


Figure 9: Gates Graphical Form Example

4. Conclusion

This paper described the design and implementation of a program visualization tool that will be useful in re-engineering and re-use of legacy code. The potential of visual programming for increasing program comprehension is being quantified through experimentation with human subjects. The project itself has given us experience in reuse as we adapt existing tools to accomplish our goals. Subsequent phases of the project will include enhancing the program browser to produce complete visual representations of C programs, and using the program browser to conduct experiments to quantify the effect of program visualization on program comprehensibility.

Acknowledgments

The work described in this paper was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

1. Baecker, Ronald M. and Marcus, Aaron. *Human Factors and Typography for More Readable Programs.* Reading: Addison-Wesley Publishing Company, 1990.

2. Basili, Victor R., "Viewing Maintenance as Reuse-Oriented Software Development", IEEE Software, 7(1), January 1990

3. Banker, Rajiv D. and Srikant M. Datar and Chris F. Kemerer and Dani Zweig, "Software Complexity and Maintenance Costs", Communications of the ACM, 36(1 1), November 1993

4. Chang, Shi-Kuo, "Visual Languages: A Tutorial and Survey," */EEE Software,* Volume 4, Number 1, January 1987, pp. 29-39.

5. Chang, Shi-Kuo, ed. *Visual Languages and Visual Programming. New* York: Plenum Press, 1990.

6. Kernighan, Brian, "PIC - A Graphical Language for Typesetting: User Manual," Computing Science Technical Report No. 116, AT&T Bell Laboratories, Murray Hill, New Jersey, May, 1991.

7. Moher, Thomas, David C. Mak, Brad Blumethal and Laura M. Leventhal", "Comparing the Comprehensibility of Textual and Graphical Programs: the Case of Petri Nets", in Empirical Studies of Programmers: Fifth Workshop, C. R. Cook, J. C. Schotz and J. C. Spohner, editors, 1993", pp. 137-161.

8. Myers, B. A. "Visual Programming, Programming by Example and Program Visualization: A Taxonomy. " In *Conference Proceedings, CHI'86: Human Factors in Computing Systems,* Boston, Mass. New York: ACM Press, April 13-17, 1986, pages 59-66.

9. Nassi, I. and Ben Shneiderman", "Flowchart Techniques for Structured Programming", SIGPLAN Notices 8(8), August, 1973

10. National Instruments Corporation. *Lab VIE W 2 Analysis VI Library Reference Manual.* Austin: National Instruments Corporation, 1990.

11. Ousterhout, John K., "Tcl and the Tk Toolkit", Addison-Wesley, 1994

12. Petre, M. and T. R. G. Green, "Learning to Read Graphics: Some Evidence that 'Seeing' an Information Display is an Acquired Skill", *Journal of Visual Languages and Computing, vol. 4, 1993,* pp. 55-70.

13. Price, Blaine A.; Baecker, Ronald M.; Small, Ian S., "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages,* Volume 4, Number 3, September 1993, pp. 211-266.

14. Singh, Gurminder and Chignell, Mark H.. "Components of the Visual Computer," *The Visual Computer,* Volume 9, Number 3, September 1992, pp. 115-142.