Qatar Univ. Sci. J. (1999) 18 : 261 - 272

DESIGN AND IMPLEMENTATION OF A SYSTEM FOR VISUAL PROGRAMMING

By
Saud M. Maghrabi
Department of Mathematical Sciences
Faculty of Applied Sciences
Umm Al-Qura University
P. O. Box 6648
Makkah, Saudi Arabia.

تصميم وتطبيق للبرمجة البصرية

سعود محمد مغربي
قسم العلوم الرياضية، كلية العلوم التطبيقية، جامعة أم القرى
صندوق بريد ٦٦٤٨، مكة المكرمة، المملكة العربية السعودية

يهدف مشروع هذا البحث الى تصميم وتطبيق نظام للبرمجة البصرية، مستندا على نموذج لتدفق البيانات. ويستند تطبيق تدفق البيانات على تحويلات جريان البيانات من خلال النموذج. يزود نظام هذا البحث المستخدم بوسيط بيني ينشأ فيه النماذج البيانية بطريقة طبيعية، وبدون فرض أي تقييد غير ضروري على مستخدم النظام، وهذا الوسيط البيني للمستخدم جذاب المنظر، مباشر في استعماله، وليس هناك تقييد في عدد النماذج البيانية أو البرامج التي تنشأ فيه أو تحرر من خلاله في نفس الوقت. وتتم محاكاة نموذج تدفق البيانات باستعمال مجموعة من التراكيب مرتبطة مع بعضها البعض بواسطة مؤشرات لتمثيل منابت واتجاهات تدفق البيانات. تم اختيار نظام هذا البحث على بعض البرامج المعقدة مثل المضروب، دوال سلسلة فيبوناكسي، وطريقة نيوتن رافسون لتقريب جذور أي دالة. وفي نهاية هذا البحث، تم عمل مقارنة بين طريقة البحث مع طرق بحوث مشابهة له. وأظهرت هذه المقارنة أن نظام هذا البحث له مميزات غير موجودة في نظم البحوث التي تمت مقارنتها مع نظام هذا البحث.

ABSTRACT

The aim of the project of this paper is to design and implement a system for visual programming, based on data flow graphs. Data flow is applicative and based on transformations on data flowing through a graph. The system provides a user interface from which the graphs can be created in a fairly natural way, without enforcing any unnecessary restrictions on the user. The interface is attractive to look at, fairly straightforward to use and there is no restriction on the number of graphs or programs that may be created or edited at once. The data flow execution model adopted is a demand driven model, which runs directly from the internal representation of the data flow graphs. The system has been tested on several complex programs including factorial and Fibonacci sequence functions, and the Newton-Raphson method of approximating the roots of a function. A comparison has been made between the system described in this paper and other related systems. It has been shown that the system of this paper has many features over the compared systems.

KEY WORDS : Algorithm on visual programming, visual programming environments, flow-graph languages.

1 Introduction

Visual programming is defined as "sets of pictures obtained by spatially arranging graphical objects over a given vocabulary through composition rules" [1]. With visual programming, there is a need for visual programming environment, which provides graphical of iconic elements that can be manipulated by the user in an interactive way according to some specific spatial grammar for program construction.

Visual programming, like much else in computer science, has a relatively short direct history. In 1963, Sutherland [2] created the first interactive visual programming language. In the early 1970s, researchers at Xerox Parc constructed the first visual programming environment. Bitmapped graphics and Mice systems can be mainly made to this research laboratory. In the 1980s, Apple computer and Sun micro-system spread graphic user interfaces to researchers and consumers. Recently, the creation of working windowing systems for PCs (Microsoft) and (X windows) has produced many systems based on graphical user interfaces.

Visual programming has attracted a great deal of recent interest. This is mainly due to reduced computer costs, affordable software and the growth of people using personal computers. The change in the type of people using computers has created a demand for software that allows non-expert users to perform specific tasks. As traditional programming techniques are often time consuming and difficult to learn, there is a great interest in providing an easier-to-learn solution. Visual programming is one attractive alternative to traditional computer languages. The motivation behind visual programming languages is based on several assumptions:

* People in general prefer working with pictorial images to text.
* Pictures can convey a more powerful message than text.
* No specialized language knowledge is necessary to understand a picture.
* Visual programs allow the expression of program interconnections and dependencies in a more natural way [3].

Visual programming allows programs to be constructed using visual elements such as icons, graphs, diagrams and pictures. Graphs are a popular general model for visual programming. There are two main classes; those based on control flow, and those based on data flow. The control flow paradigm supports procedural programming and relays heavily on design notations, especially flowcharts and Jackson Structure Program diagrams [4]. Visual programming accept two notions of flow of control in

program: imperative and declarative. With the imperative approach, a visual program constitutes one or more control flow which indicate how the thread of control flows through the program. With the declarative approach, one only needs to concern with the computations that are performed, and not how the actual operations are carried out.

The data flow paradigm is based on the data flow model. In its simplest form, a data flow program is represented by a directed graph. Arcs passing into nodes represent input data, the nodes indicate transformations on the data, and arcs flowing out of a node carry output data. The data flow paradigm is oriented to flow-type operations. Objects of data are shown in relationship to procedures. No decision logic is shown; the data flow paradigm is used to model the flow of data. Because this data flow model contains no conditional or iteration constructs, and no order of node evaluation is specified, the system of this paper uses the data flow paradigm. Therefore, the user of this system does not require to keep track of how sequencing of operations modifies the state of the program. Moreover, features, include functional abstraction and control constructs, can be used to augment this simple data flow model in visual programming.

The aim of this research is to design and implement a system for visual programming based on data flow graphs. A program, written in C, has implemented the system, and it uses the X-windows graphical interface package. The potential of the approach is demonstrated using several complex test programs including factorial and Fibonacci sequence functions, and the Newton-Raphson method of approximating the roots of a function.

The paper starts with a description of the internal representation, showing the main data structures needed for building the system. The main data structures are graph, nodes, arcs, and ports. After showing how the main data structures are built, the system components are described. The system consists of an interface and an execution model. They are described through an example, to show the steps that a user takes in constructing and executing a graph. Next, a comparison between the present system and some recent existing systems is given. Finally, a conclusion and further research are discussed.

2 Internal Representation

A visual graph built by the system's user is represented internally using arrays of pointers. The main structures (graphs, nodes, arcs, and ports) are collected together when they are created, by putting pointers to them into an appropriate array (see figure 1). Relationships between structures are realized using pointers.

Graph structures allow a whole graph to be encapsulated into one structure, making identification and manipulation of specific graphs possible. A graph structure is created when the 'new graph' button is clicked on the palette, taking its name from the palette text field. It contains arrays of pointers to the constituent nodes, ports and arcs.

There are three kinds of nodes: operators, literal values, and calls to other graphs. Each node structure contains a pointer to an evaluation function. The different types of node are differentiated because they point to different evaluation functions. When a palette button is dragged and dropped, a call is made to a function to create and place the node, passing it a special token according to which kind of node is being placed. The function contains a switch statement, that controls which shape and label are to be drawn and which evaluation function will be associated with the new node structure.

Arcs structures fulfill two main purposes: to manage the transfer of data from one node to another, and to support graphical drawing of lines when building the graph. The ability to route lines is essential for building clear meaningful graphs. The arc structure can therefore store several straight segments, kept in an array. When an arc drawing action is started, the initial coordinates are placed in an array. When a line is rubber banded and the button is

released, the position of the end determines whether or not the user is routing the line. If the end of the line is some distance from any port, its coordinates are placed in the array and a flag is set so that next time the user starts to draw a line, it starts from this last position (regardless of where the cursor is). Then, the same process is repeated so that the arc is made up of segments with their ends stored in the array.

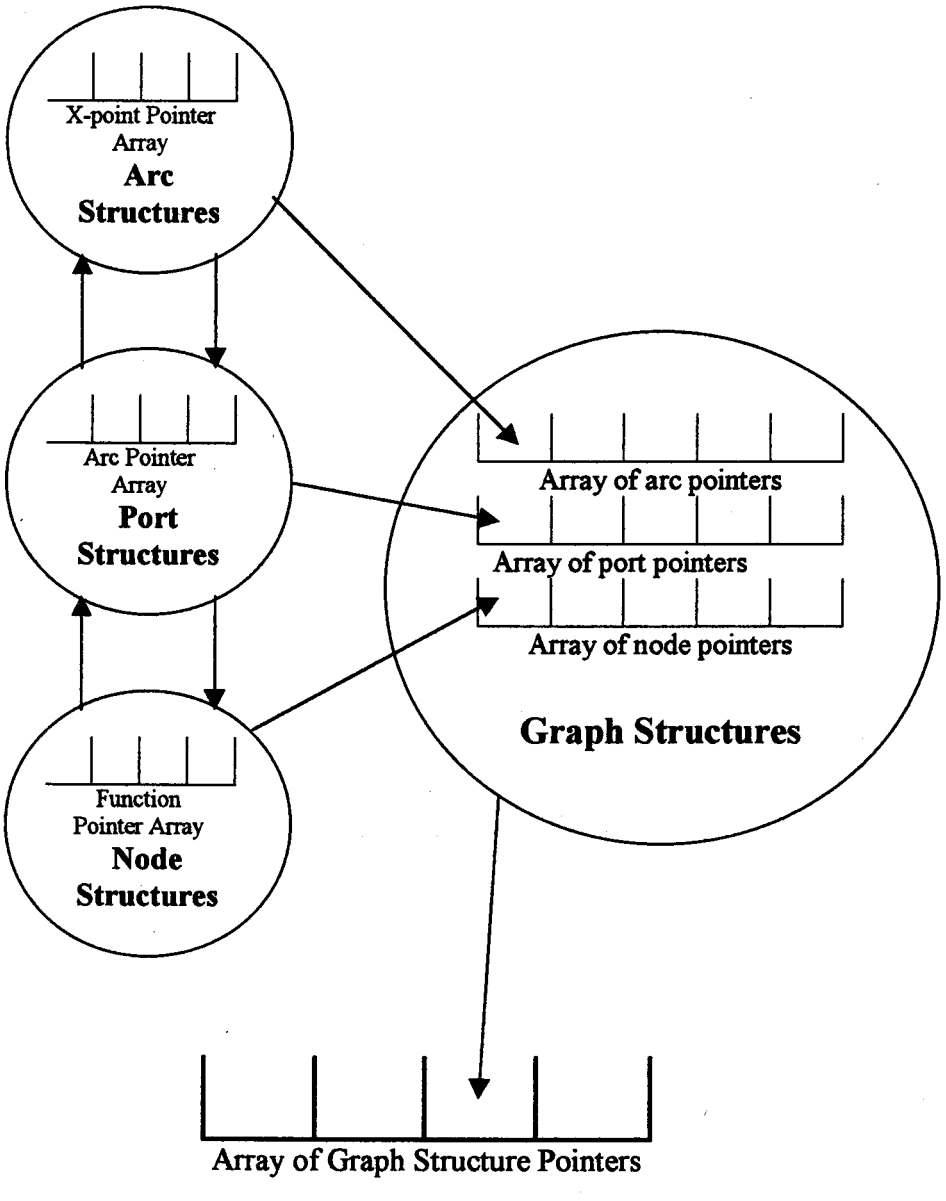

Figure 1. Diagram expressing the internal representation of different components of the system.

The port structure represents the inputs and outputs to a node. It contains a pointer to the node it is attached to. This provides a way of reaching the node from the port, necessary in the execution of the graph. It also contains an array of pointers to the arcs that connect to this port. A tally of the number of lines connected needs to be kept for evaluation purposes.

3 System Components

The system encompasses two main areas: user interface, and execution model.

3.1 User Interface

The initial view of the system is the palette. The palette consists of sliding blocks of operator buttons, grouped together in related sets. The operators on the palette are grouped under labels, which describe their functions.

From the palette multiple editing areas can be created, each served by the functions on the palette. The user can create any number of graphs, with different names. A new graph can be created by entering the name of the graph in a text-input box, and clicking on a 'new graph' button. When a new graph is created a corresponding button, with the same name, is created in the 'Graphs' pane of the palette. This can be then clicked to pop up an editing area for the graph.

The construction of a graph to recursively calculates the factorial of its input is used as an example to see how a graph built by the system is used. This will show the steps that the programmer takes in constructing a graph, and it also explains what happens internally as the graph is built and executed. The system initially presents the user with the palette. The graph name, factorial, is entered in the text field and new graph button is pressed. The user is presented with a new button that can be clicked on to open the editing area of the factorial graph. At this stage the system will appear as in Figure 2.

The next step is to place the input. The input can be named using the palette text field, then dragged and dropped into the editing area. As the input box is placed, externally the on screen representation is drawn. The input box appears as a box containing two components: a number corresponding to the input port position when the graph is called, and its name. Internally, a general node structure is used to represent the input, with a node type-specific evaluation function pointed to from within it. Next, any ports related to that node are created and drawn. The pointers of the newly created port and node are placed into the corresponding arrays holding the components of the factorial graph. The count of the number of inputs to this graph will be incremented to one.

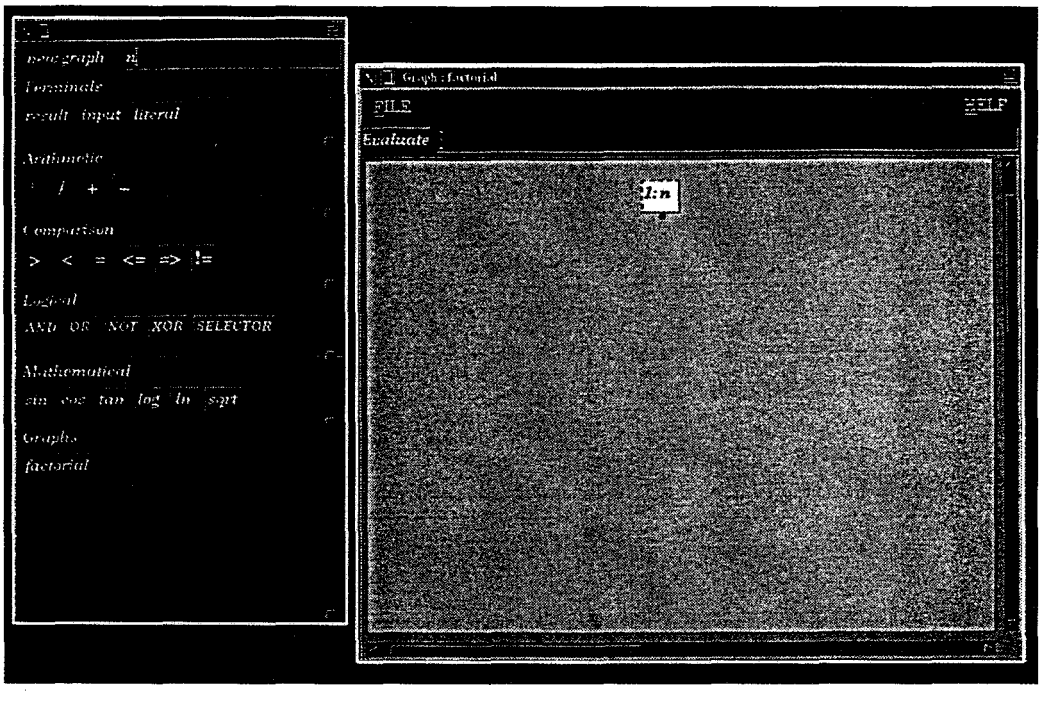

Figure 2 : Palette and Editing Area for factorial's graph.

Literal inputs to the graph can now be added. To place the literal nodes, enter the value in the text field, click on the literal button and drag and drop it as before. Internally the node is represented by the same structure as before, but with a different evaluation function. The string entered in the palette text field is extracted and used to derive a value for the literal; this is then stored within the node. The other operator nodes can then be clicked on and dragged across into the drawing area. The construction of the internal representation for these nodes is done in a similar way as before.

To represent the recursive call of the factorial graph to itself, the factorial button is dragged and dropped onto its own graph. The graph node dropped is rectangular to distinguish it from other operators. It has the same number of input ports, as there are inputs to the graph, that is, one. The graph node is represented by the same structure, but with a specific evaluation function attached. The width of the rectangle is calculated to fit the graph name. The system now looks as in Figure 3.

To link all the nodes together to represent the flow of data around the graph, click at a desired starting port and route the line to its destination. As an arc is connected to the initial port, internally an arc structure is created; this contains information on the drawn line segments and also the two ports connected.

The graph is now complete and it realizes a recursive factorial function, as shown in Figure 4. It is now necessary to build a graph to test out the function. This graph is constructed visually and internally in exactly the same way, but in this graph there is no input box as its only purpose is to define what literal value will be supplied as input to the factorial function

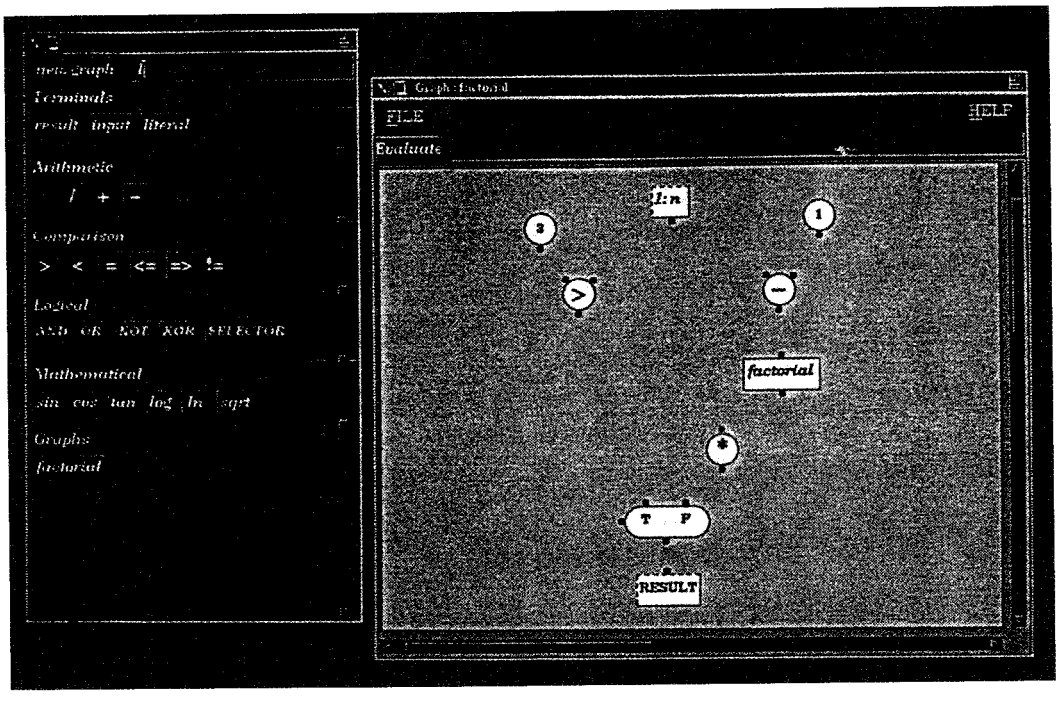

Figure 3 : Nodes placed on factorial's graph.

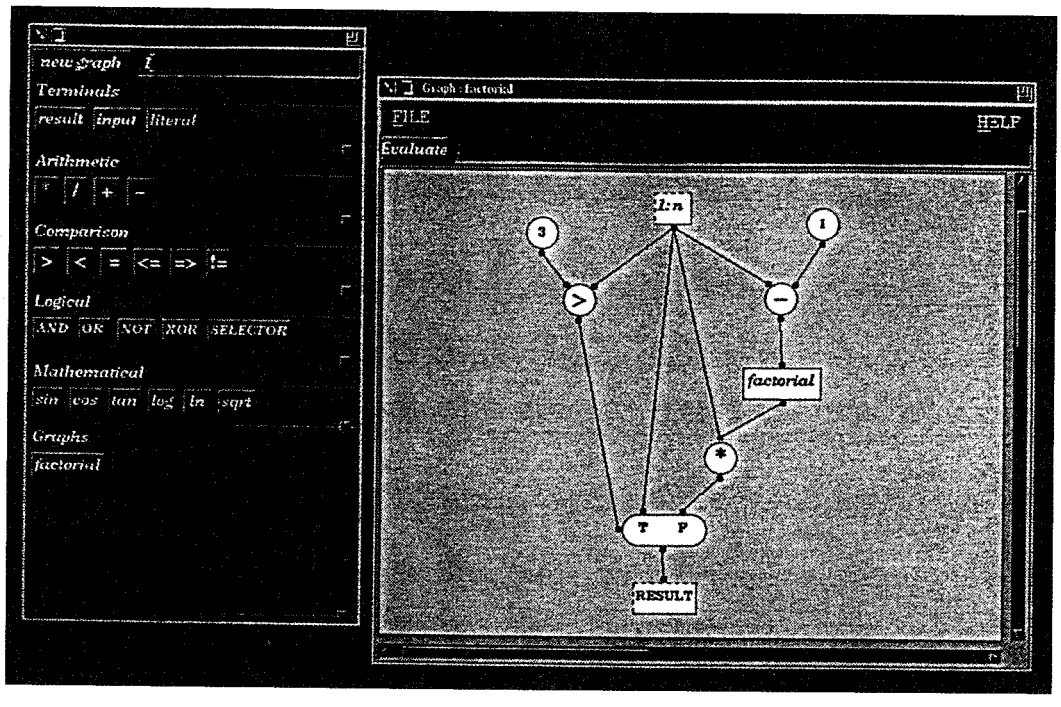

Figure 4 : Complete factorial graph.

3. 2 Execution Model

An execution model is needed to define the way in which data flow graphs are interpreted. A demand driven execution model has been used, in which data flows downstream but there is a complementary notion of 'demands', which flow upstream. A node fires when a demand arrives at it from downstream. During its own evaluation, the node sends a demand out on an input arc when data is required for that input. Execution starts when the 'evaluate' button is clicked in a graph-editing window. The result box node in that graph is evaluated and the result displayed in the text field.

The system uses a selection mechanism, allowing routing of the data flow graph. A selector is a way for controlling the path of data in a data flow graph. It has two data inputs and a Boolean input; one of the input data values is placed on the output arc, depending on the state of the Boolean control input (see figure 5). The following selector evaluation function *evalutSelector* is used.

```
data evalutSelector(node-ptr n)
{
    data result, cond;
    cond = inputvalue(n,0,0);
    if (cond.type!=BOOL) {
        printf('wrong type need boolean\n");
        }
    if (cond.value.b==TRUE) {
        result = inputvalue(n,l,0);
        }
    else {
        result = inputvalue(n,2,0);
        }
    return result;
};
```

It first demands the Boolean control input using a call to *inputvalue*. Once this has been evaluated, one of the inputs to the selector will be demanded. The other input will not be demanded. This is an example of the major property of demand driven execution: it only demands the inputs that are required, giving more efficient execution. In this code, input 0 is the Boolean control, input 1 is the true input and input 2 is the false input; it can be seen from the function that inputvalue is called for either the true or false input, but not both.

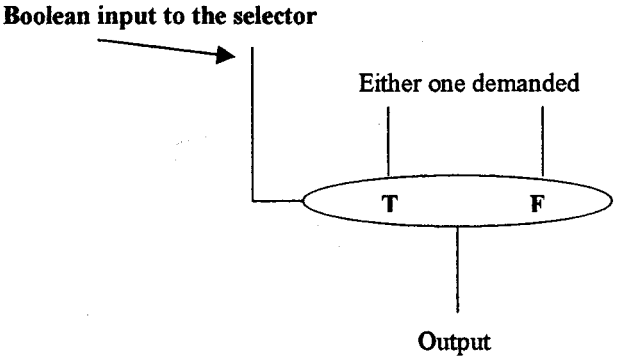

Figure 5: Diagram showing the architecture of the selector.

In the factorial example, the selector controls the recursive base case. When the input is greater than two the recursive calls to factorial are made because the 'false' input to the selector is demanded, and this propagates to the recursive call. But if not, the 'true' input is demanded and this just returns the output value, terminating the graph recursion. Literal nodes terminate the recursive execution algorithm, since they have no inputs to demand. The graph can now be evaluated by clicking on the evaluate button at the top of the editing window. Now the execution cycle is initiated on the internal graph structures, as shown in Figure 6.

4 Implementation

The system has been implemented as a C program. The program seems to fall naturally into three main areas. These areas are the implementation of the interface, the internal representation of the graph, and the implementation of the execution model. The program uses the X-windows graphical interfaces package. The X-window system is a user interface environment for engineering workstations [5-7]. It consists of three main levels:

1. Xlib contains a library of over 300 C functions that responding to user interface events and drawing graphical output to the display.
2. The Xt Toolkit is a higher level programming interface to Xlib and supports the use of user interface components called widgets.
3. Widget sets are implemented with calls to the Xlib and Xt Toolkit libraries.

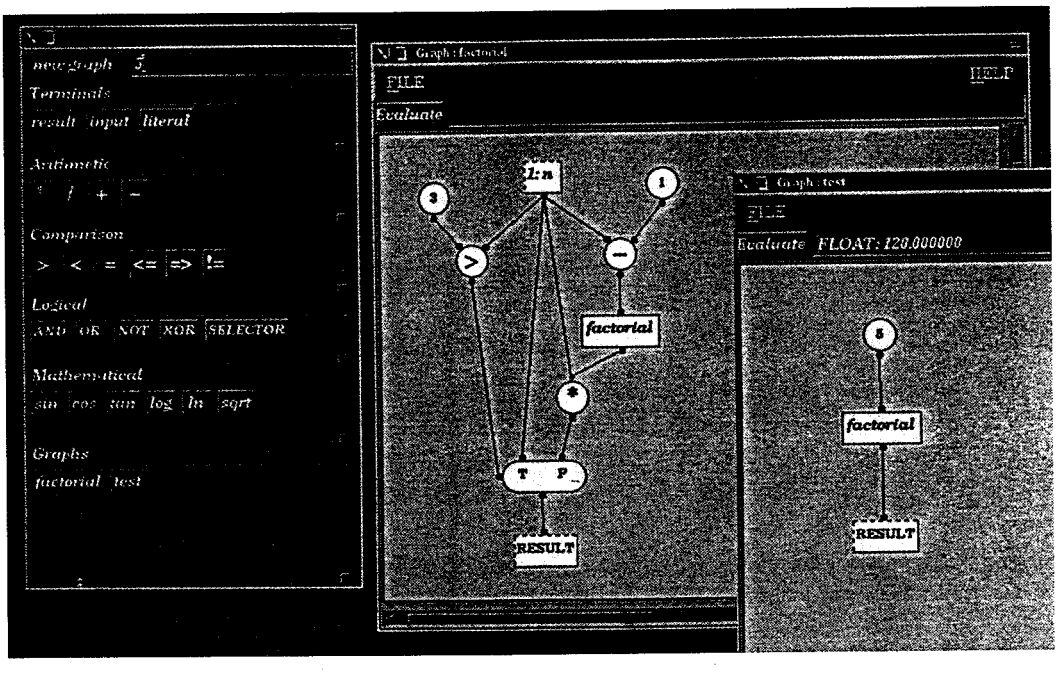

Figure 6 : Factorial graph and test graph.

The program is basically controlled from the interface code. Within the interface, it is necessary for some widgets to perform some form of action when an event such as a mouse click occurred. The following library routines are invoked in the presented system:

XsetWindowBackground(),
XsetForeground(),
Xdrawline(),
XcopyArea(),
XfillArc(),
Xsetfont(),
XtextWidth(),
Xdrawstring(),
Xstorecolor(),
Xcreatepixmap(),
Xfillrectangle(),
SetXcolor(),
Xtaddcallback(),
Xtwindow(),
Xtwindowtowidget(),
Xtgetvalues(),
Xtdispatchevent(),
Xtappnextevent().

Figure 7 illustrates an example, which has been produced by the program to illustrate the usefulness of the system to address practical problems. It shows the graph for the absolute function with the following features: evaluating of the function, finding the root of the evaluation, computing new approximation to the function, showing converging of the function, accepting testing values, giving the first derivative to the function.

5 Related Systems

I compare related works with the system described in this paper through the lens of a taxonomy of the area given by Singh and Chignell [8]. Sigh and Chignell [8] divide visual computing into three main categories: visual aids for programming, end-user interaction with computer, and visualization. This taxonomy is based on the viewpoint of the user of the computer. The first kind of the taxonomy is important for program designer, and the second is the main concern of the user interface. Users, who have to interpret and process large amounts of data, are interested in the third kind of the taxonomy. The main focus of this paper lies within the first category of the taxonomy. The remaining two kinds of the taxonomy are ignored in this paper.

Microsoft's Visual Basic® and Visual C++® could be placed within the 'end-user interaction' group of Singh and Chignell's categorizing. In these window applications, interactive screens are easily generated. This can be done by choosing an icon (either a standard icon or a user-generated icon), placing it in the desired screen position, setting the parameters, then programming textually what events will occur when that icon is chosen during execution. Although these languages greatly simplify the task of building window applications, they do not provide a facility for algorithm animation. The method described in this paper provides this facility.

The methods described in references [1] and [9] are for constructing electric circuits and a condition IF. The method of reference [1] generates a data flow graph either as a simple argument, or as an operation taking inputs from the outputs of two data flow graphs, or as a multiplexer taking as inputs the outputs of two data flow graphs and of a condition IF [1]. The system of reference [10] constructs sets of diagrams from primitive components to visually compute logical notations. The system of this paper provides a powerful programming tool for solving many mathematical problems. It is based on data flow graphs. Data flow is based on transformations on data flowing through the graph. Five attributes can be established for a qualitative comparison between the present method and other methods [1] and [9] (see Table 1).

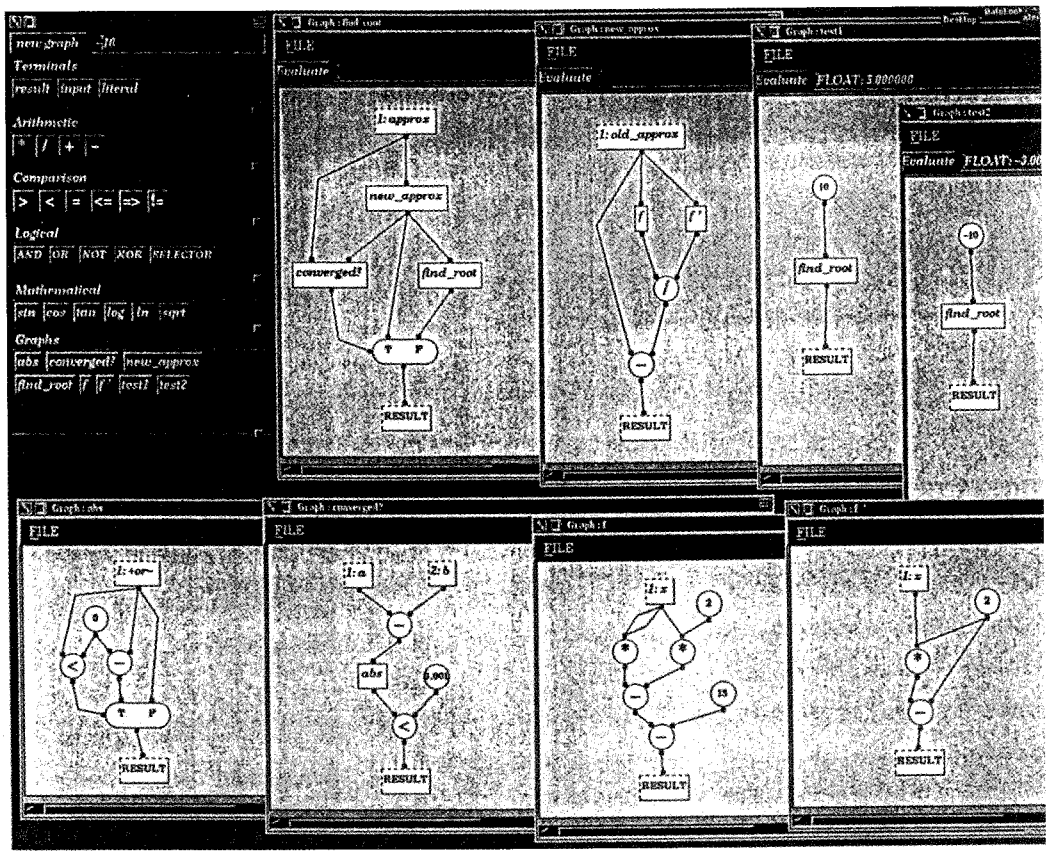

Figure 7 : An example showing how the system handles practical problems.

From table 1, the present method has many features over other systems, including functional abstraction, control construction. In the method described in this paper, functional abstraction allows an entire graph to be considered as a function, so that a single node in another graph can represent it. Arcs going into the node may then be considered arguments to the function. Functional abstraction is important if graphs are to represent non-trivial programs. It is also necessary to support repetition through recursion. Recursion allows data arcs to carry sets of elements, and functions themselves, as items of data.

The method of this paper provides a graphical environment for the construction of visual program graphs, and maintains an internal representation that enables their execution. It has the visual features used in the visual programming challenge [11-12], such as explicit representations, contextual information, and execution animation.

5 Conclusion and future work

There is a move towards greater exploitation of pictorial images as an aid to traditional program understanding and construction, as well as great interest in alternative forms of program representation. This paper describes a system for visual programming, based on data flow graphs. Data flow is based on transformations on data flowing through the graph. The implementation of the system has been achieved by using the *C* language and X-windows to provide a graphical environment. The system has been used to construct and execute data flow programs of varying complexity. These test programs include factorial and Fibonacci functions, and an eight-graph program to find the roots of a function using Newton-Raphson's method.

The system of this paper could be extended by adding a selector to the data flow graph to provide "control structure" based on lazy evaluation. Before a node is evaluated, a check is made in the frame of node values at

the position of this node, to see whether a value has already been computed. This is done by adding a selector to check if the value is still as it was when created. Another way to extend the system of this paper is to make it solve more general-purpose problems. It would be useful if functions could be passed through the graph as values, rather like LISP. This could be used to implement more sophisticated forms of control, for example, functions could be used to iterate over sets of data.

Table 1 : Comparison of visual language attributes for the system described in this paper and the systems described in references [1] and [9].

Attribute	The system of this paper	The system of reference [1]	The system of reference [9]
Support for data encapsulation	Strong	Moderate	Weak
Support for one paradigm	Strong	Weak	Weak
Multi-paradigm support	Data flow combined with object-oriented programming	One only (data flow)	One only (data flow)
Type of graphics	Meaningful icons and diagram	Less meaningful icons and diagram	Fairly meaningful icons and diagram
Functional abstraction support	Strong	None	None
Control construction support	Strong	None	None
Functionality	Applicable to mathematical area	Applicable to electric circuits and a condition IF	Applicable to electric circuits and a condition IF

REFERENCES

[1] **Costagliola, G.,De Lucia, A., and Tortora, G., 1997.** "A Parsing Methodology for the Implementation of Visual Systems", *IEEE Transaction on Software Engineering*, Vol. 23, No. 12, 777-799.

[2] **Sutherland, I. B., 1963.** "SKETCHPAD, A Man - machine Graphical Communication System", *Proceedings of the Spring Joint Computer Conference*, 329-346.

[3] **Hils, D. D., 1992.** "Visual languages and computing survey: Data flow visual programming languages", *Journal of Visual Languages and Computing*, Vol. 2, No. 3, 69-101.

[4] **Jackson, M. A., 1975.** Principle of Program Design, Academic Press.

[5] **Culter, E., Gilly, D., and O'Reilly, T., 1992.** The X-window System in a Nutshell, O'Reilly & Associates, Inc, Second Edition.

[6] **Mikes, S., 1991.** X Window Program Design and Development, Addison-Wesley Publisher.

[7] **Parlidis, T., 1999.** Fundamentals of X programming: Graphical User Interface and Beyond (Plenum Series in Computer Science), Kluwer Academic Publishers.

[8] **Singh, G. and Chignell, M. H., 1992.** "Components of the Visual Computer", *The Visual Computer*, Vol. 9, 115-142.

[9] **Costagliola, G., Tortora, G., Orefice, S., and De Lucia, A., 1995.** "Automatic Generation of Visual Programming Environments", *IEEE Computer*, Vol. 28, No. 3, 56-66.

[10] **Agusti, J., Puigsegur, J., and Robertson, D., 1998.** "A Visual Syntax for Logic and Logic Programming", *Journal of Visual Languages and Computing*, Vol. 9, No. 4, 399-427.

[11] **Citrin, W., Ghiasi, S., Zorn, B., 1998.** "VIPR and the Visual Programming Challenge", *Journal of Visual Languages and Computing*, Vol. 9, No. 2, 241-258.

[12] **Heger, N., Cypher, A., and Smith, D., 1998.** "Coca at the Visual Programming Challenge", *Journal of Visual Languages and Computing*, Vol. 9, No. 2, 151-169.