Report Number: WUCS-93-53

1993

Completeness of a Visual Computation Model

Timothy B. Brown

Visual programming is the specification of computational processes using diagrams and icons. Traditional computation models such as Turing machines and lambda-calculus, which are based on one-dimensional text strings, are not suitable for visual programming languages. We propose a two-dimensional computation model that requires no text. We also prove that the model is computationallhy complete, i.e., that the model has the same computational power as Turing machines.

Completeness of a Visual Computation Model

Timothy B. Brown

WUCS-93-53

November 1993

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

A thesis presented to the Sever Institute of Washington University in partial fulfillment of the requirements for the degree of Master of Science, December 1993. Prepared under the direction of Professor T. D. Kimura

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

COMPLETENESS OF A VISUAL COMPUTATION MODEL

by

Timothy B. Brown

Prepared under the direction of Professor T. D. Kimura

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE

December, 1993

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY

ABSTRACT

COMPLETENESS OF A VISUAL COMPUTATION MODEL

by Timothy B. Brown

ADVISOR: Professor T. D. Kimura

December, 1993

Saint Louis, Missouri

Visual programming is the specification of computational processes using diagrams and icons. Traditional computation models such as Turing machines and λ-calculus, which are based on one-dimensional text strings, are not suitable for visual programming languages. We propose a two-dimensional computation model that requires no text. We also prove that the model is computationally complete, i.e., that the model has the same computational power as Turing machines.

Table of Contents

List of Tables

List of Figures

Figures Page

Figures Page

1. Introduction and Problem Statement

Visual programming languages allow the specification of computational processes using visual expressions. These visual expressions are usually in the form of two-dimensional diagrams and icons. The current trend in visual programming language research is towards the creation of sophisticated, general purpose visual programming languages. This is a natural and useful trend. However, we believe that in order for the power of visual programming languages to be realized, the fundamental aspects of visual programming languages must be discovered.

One of those fundamental aspects is a formal computational model which can serve as a basis for visual computation. Since traditional computation models such as Turing machines and λ-calculus are based on one-dimensional text strings, they are not suitable for visual programming languages. We propose a two-dimensional computation model that requires no text. Our proposed model is visually expressed using only boxes, arrows, and a few notations for primitive operations. The model is also computationally complete, i.e., it has the same computational power as Turing machines.

In creating our computation model, we have adopted as a guiding principle Einstein's often quoted advice, "Make it as simple as possible, but not simpler." Following this advice, we attempt to define and implement the simplest visual programming language ("as simple as possible") which is computationally complete ("but not simpler"). We do not claim that the proposed model is an ideal visual programming language, any more than one would claim that the Turing machine model is an ideal text-based programming environment.

Before presenting our visual computation model, we provide some background information about visual programming languages in general and about recent trends in visual programming systems. After this background material is presented, along with further motivation for this research, we describe the methods and metrics chosen for the research. The visual computation model is then described. A proof of the model's computational completeness follows the description of the model. Our current implementation of the model is then described. Lastly, an evaluation of the work and some suggestions for future improvements are provided.

2. Background and Motivation

2.1 Introduction to Visual Programming Languages

Traditionally, computational processes are specified by one-dimensional text strings. Visual programming, in contrast, uses two-dimensional diagrams and icons for specifying such processes. For many users, visual programming is an attractive alternative to traditional computer programming. One of the reasons for this attractiveness is that the visual representation of a solution to a problem is often more closely matched with the way the solution is conceived or understood than is a text-based representation. The evidence for this assertion can be found in many places.

For many people, spoken or written directions about how to get from one place to another are much more difficult to understand than a simple map. Similarly, instructions for procedures to be carried out in a work environment, such as a factory or assembly line, are often much easier to understand when they include pictures showing what is to be done. Instructions for how to work with computer software commonly include not only pictures and diagrams of what the user should see on screen, but also diagrams and pictures that help the user build a conceptual framework for the workings of the software. Even people who are supposedly very practiced at representing problem solutions in text-based languages, namely computer programmers, commonly use diagrams to describe their solutions both before and after programs are written.

Traditional, text-based, programming languages are notorious for the difficulty many people have in using them. As one researcher puts it, "The facilities these [text based] languages provide for describing algorithms correspond more closely to how computers operate than to the cognitive or perceptual processes of the programmer." [9]

In response to this gap between cognitive processes and computer processes, graphical aids have been used for program design for many years. The most widely known example would have to be the flow-chart, but it is certainly not the only example. Diagrams showing the relationships among subroutines and their calling routines were commonplace almost as soon as the idea of subroutines was conceived. The documentation of any program written in a modular programming language would be incomplete without a diagram showing the relationships among modules. Similarly, documentation of object-oriented systems inevitably includes diagrams showing the class/subclass (object/extension) relationship among objects in the system. This is so common and useful that an object-oriented programming system that doesn't include a "class browser" which can dynamically create and display such diagrams and allow the programmer easy access to the code for any object through interaction with those diagrams is considered an inadequate object-oriented environment.

Two-dimensional notations and languages have also been used freely in mathematics and symbolic logic. Obvious examples of these notations are graphs and trees used to define and illustrate relationships between objects. Another well-known example is the Venn diagram. Some less well-known examples include Frege's [11] use of notations based on binary trees to represent structures of various judgments. Peirce's system of existential graphs [35] is a diagrammatic system for first order logic. More recently, Harel [18] presented a visual notation which is an extension of Venn diagrams to include the ability to represent relationships commonly represented in graphs.

The idea of allowing a computer to do the work of transforming a diagram used in program design into a working program is actually not very new. Researchers at M.I.T. were exploring interactive programming by means of flowcharts on a graphics display as early as the late 1960's [37]. But it was not until the cost of graphics-related hardware and software began to fall significantly that the visual programming idea gained significant momentum. Even with graphical computer systems commonplace today, the potential advantages of visual programming have not been fully appreciated. We believe that this is caused, in part, by the hardware most commonly used to interact with graphical computer systems, the mouse.

Using a mouse is not a natural skill, the distance between where an action takes place, at the mouse, and where the results of that action are displayed, on screen, is somewhat troublesome. This makes creating diagrams with a mouse rather awkward, difficult, and time consuming. Recent advances in pen-based interfaces have changed the situation. Even fairly experienced mouse users can complete diagrams more quickly and easily using a pen-based interface [4]. The use of shape recognition algorithms and pen-based systems makes the drawing and composition of diagrams for visual programs easier and more natural [5]. The advent of pen-based personal computer systems will almost certainly create more demand for systems which allow complete interaction using only the pen. These systems will include pen-based visual programming languages.

A wide variety of visual programming languages have been proposed, developed, or commercialized [13]. Researchers in the area have devised a number of methods of classifying such languages.* We will not attempt to present here all the different classification information about a given visual language. However, we will provide some basic information using a few of the classification methods presented in Hils [19], Glinert [13] and Ambler [3].

One of the very first design questions a visual programming language creator must answer is what will be the *semantic base* of the language. One of the available

* A method of classifying all programming languages has been proposed which parallels the Flynn Classification of computer systems: Single Instruction Single Data, Single Instruction Multiple Data, Multiple Instruction Single Data, and Multiple Instruction Multiple Data. In this scheme, languages are classified with regard to their visual content. The four classifications are: Textual Instruction Textual Data (TITD), Textual Instruction Visual Data (TIVD), Visual Instruction Textual Data (VITD), and Visual Instruction Visual Data (VIVD). All the languages discussed in this thesis have at least some form of Visual Instruction.

choices is the flow paradigm. The flow paradigm can be divided into two instances, data flow and control flow. In a data flow system the sequence in which functions or operations in a program are to execute is not explicitly specified. Instead, the source of data for an operation is specified, and whenever all the inputs for an operation are available, the operation occurs. The operation is said to *fire* whenever its data is available. The control flow model more closely resembles the traditional use of flow charts. Whatever visual representation is used shows the sequential relationship among operations. Some data flow based systems also include control flow constructs.

An alternative is to create a constraint based system. Such systems allow the user to visually specify the invariant properties and relationships of objects in a particular problem space along with some methods which the system may used to maintain those relationships. Such systems are in many ways analogous to the textual programming language Prolog and other logic programming systems [7, 47]. There are also visual languages which use programming by demonstration or rehearsal. In such systems, the user manipulates sample data or visual representations of data structures to demonstrate to the system what operations are to be performed. The system then emulates the demonstrated operations on new data.

The last major class of visual programming languages is the group of forms-based systems. Ambler and Burnett, leading researchers in this visual programming paradigm, have stated, "You can think of form-based programming as a generalization of spreadsheet programming."[3] The basic idea is that the success of spreadsheets is largely due to the visual methods used within spreadsheets to represent relationships between data items and that visualness can be expanded upon to create more powerful problem solving systems.

Once a semantic base is chosen, a *syntactic base* must be chosen. The syntactic base determines the actual visual representation of programs. In flow based systems the most common approach is to use directed graphs. Nodes in the graphs indicate operations or data cells and the directed edges represent the flow of either the data or control. Some flow based visual languages, for example HI-VISUAL [20], use juxtaposition of nodes to indicate flow. Nodes in the graph are represented as boxes, icons, or other shapes. This visual representation has been called the box-line representation or 'boxes-on-lines'. Lieberman [28] has suggested that the predominance of this visual representation is a hindrance to the imaginative use of visualization in programming language systems. Constraint based systems often use box-line representations or domain-specific representations, for example circuit diagrams or matrix notation. Since forms based systems are extensions of spreadsheets, their visual representation generally looks like a spreadsheet interface. The visual representation and syntax of a demonstration system are usually domain specific.

After the semantic and syntactic bases are chosen, the language designer then creates a set of *basic constructs*. These constructs include such items as representations for iteration, sequential execution (in a non-control flow system), procedure abstraction and recursion, type checking, and higher-order functions.

The following section includes samples of a number of visual programming languages. Each language description includes an indication of the semantic base, syntactic base, and basic constructs of the language. The languages are presented chronologically by date of the publication.

2.2 Sampling of Visual Programming Languages

2.2.1 ThingLab (1979)

ThingLab was the product of Alan Borning's Ph.D. Dissertation [6]. It was originally designed as an environment for graphic simulations of experiments in physics and geometry. But it is now clear that one of ThingLab's major contributions to computer science is the constraint based visual programming language incorporated into the system. This programming language is an extension of Smalltalk [21,15,16] and is, therefore, quite naturally, very object oriented.

In a constraint based system, the user specifies a set of relationships, called constraints, between objects in the problem space. The system's number one job is to maintain these relationships. A graphic user interface is incorporated into ThingLab which allows users to view and edit objects. As the objects are being edited, the system is using the construction of objects as a way of determining constraints. In order to do this, the system must be supplied with some information about what constraints are possible. This is done by a different class of user, the kit-builder.

The kit-builder creates objects, by programming in Smalltalk, which are available for use by the other system users. When the kit-builder creates constraints, she must also specify a set of methods that the system can invoke to satisfy the constraints. Figure 2.2.1.1, which is taken from Borning [7], shows the result of a user creating a new Quadrilateral object using basic geometric objects (like line and point) created by a kit-builder and then adding mid-point objects to the lines and connecting those mid-points. Moving one of the end-points of the quadrilateral lines, demonstrates the theorem that the lines connecting the mid-points always form a parallelogram. When the prototype quadrilateral was created, the system developed the following description of a quadrilateral.

```
Class Quadrilateral
        Part Descriptions
                part1: a line
                part2: a line
                part3: a line
                part4: a line
        Merges
                part2 point2 = part3 point1
                part1 point1 = part4 point2
                part3 point2 = part4 point1
                part1 point2 = part2 point1
```

The Merges section shows the constraints which the system created based on the user's drawing of the prototype.

ThingLab also includes a set of objects which have constraints allowing them to perform simple mathematical operations. Figure 2.2.1.2, which also comes from Borning [7], shows a Celsius-to-Fahrenheit Converter. Notice that items that cannot be changed in order to satisfy constraints are shown with an anchor attached. Another feature of the program is that it takes advantage of the declarative (non-procedural) nature of constraints to allow conversion both forwards and backwards. In other words, the user could change the temperature on the Fahrenheit side and have the converted Celsius temperature show up on the other side, or the user could change the Celsius side and have the appropriate Fahrenheit value show up on the other side.

ThingLab was later extended to allow the constructor of basic building blocks, to do so in a more intuitive and graphical way. Instead of programming in Smalltalk, the kit-builder programs the system using diagrams which are very similar to those in Figure 2.2.1.1. Figure 2.2.1.3, which is from Borning [8], shows the program which defines the MidPointLine object and the program defining a vertical line. This system forms the abstraction mechanism allowing ThingLab's programming language to be used to define new objects, functions and procedures, using existing ones. This allows recursion to be represented.

2.2.2 Rehearsal World (1984)

One of the most well known visual programming by demonstration systems is Rehearsal World [17]. It was designed to allow teachers who do not know how to program to create computerized lessons. The system uses a theater metaphor in which the screen is the stage and there are predefined performers which a user can direct to create a play. The underlying representation of all the players is created using Smalltalk code. New performers can often be created by examples, but it is sometimes necessary to write Smalltalk code. The teachers are not expected to program in Smalltalk, therefore, some coding must be done by others.

Performers are represented on screen by icons and interact by sending cues. The director, the teacher, teaches the performers how to behave during a given production. Figure 2.2.2.1 shows how performers are grouped on screen into troupes. Some typical predefined performers are a Text performer that can be directed to display text and a Number performer that can be directed to display numbers. Each performer has a set of cues to which it will react. The user can send cues or, by telling the system to "watch", can direct a performer to send cues to other performers. The performers correspond to Smalltalk objects and the cues correspond to messages. Once the performance has been choreographed by the director, it can be rehearsed and debugged before it is played back for its intended audience.

2.2.3 Pict (1984)

One of the pioneer efforts in flow based visual programming languages was Pict [12, 14]. Pict's designers chose the flow based model, and in particular a control flow based model, even when many professionals had been questioning the usefulness of flowcharts. They contend that much of the criticism of flowcharts stems from the fact that the flowchart is *not* the program. They found "much to recommend the flowchart when it is the program itself rather than merely an aid to documenting it" [14]. The syntactic base for Pict is the directed graph. Pict systems incorporate an interactive editor allowing users to create programs using a joystick. Once the system is started, users need never touch a keyboard. Pict programs can perform simple, integer-based calculations. Figure 2.2.3.1, from [14], shows a series of displays from a running Pict system.

Since Pict uses color to differentiate many of its significant structures, the figure loses quite a bit in its translation into black and white. Nonetheless, the flowchart-like structure of Pict programs can easily be seen. Nodes in the graph are represented as small squares; arcs, or flows between the nodes, are represented as double lines with arrow heads inside indicating the direction of flow. Image (d) of the figure shows the representation of a routine to perform natural number multiplication by repeated addition; the icon for this routine is shown enlarged in image (c). This shows Pict's procedure abstraction construct. The diagram in image (e) shows a program which retrieves an input, *n,* from the user and then calls a factorial subprogram to calculate *n*!. Images (h) through (k) show the animation of the execution of the program, which uses recursion to calculate 7!. Along with procedural abstraction and recursion, Pict contains constructs, many of them represented using color, for Pascal-like control structures like *repeat-until* and *while* . Different parameter passing modes, data structures, and program operations are represented by various icons.

Experiments conducted by Pict's authors showed that naive users were very impressed that they could program a computer 'without learning a programming language.' Of course, they did learn and use a programming language, along with a program editor. They simply were unaware of their accomplishment, presumably because it was much easier than expected. Pict's authors state that they do not claim that Pict is a general purpose programming environment, but Pict clearly points toward the development of such systems.

2.2.4 Prograph (1985)

Prograph was initially reported on in 1985 [30] and is described in more detail in later publications [9,10]. Prograph is a commercially available product. One of the major features of Prograph is the integration of data flow based visual programming with object-oriented programming techniques. Classes, which are represented as icons, can be viewed in a class browser. Figure 2.2.4.1(a) shows a class browser window containing a forest of class trees. Part (b) of the same figure shows the visual representation of the attributes of some classes. A method for a class is a data flow diagram which may be condensed into an icon. Part (c) shows the representation of two of the methods of the class Index.

Prograph includes predefined iteration and parallelism mechanisms, including operators which apply a function to all objects in a list and return a list, and a control flow construct -- the *while* loop. The Prograph system includes an editor for creating classes and programs, an interpreter for executing and debugging, and an "Application Builder" which is used for constructing and testing the user interface to a program. The Application Builder is similar in concept to the techniques used for interface construction in such products as Visual Basic.

2.2.5 Show and Tell (1986)

The Show and Tell Language (STL) system is a data flow based, general purpose programming language originally designed for school children. [26, 27] As in most data flow languages, the directed graph serves as the semantic base. Boxes represent functions, constants, variables, iterators, or containers of inconsistency. Arrows represent the flow of data between boxes. The particular case of directed graph used in STL is called a *boxgraph.* In the boxgraph notation, boxes may be hierarchically nested and edges can flow from a box at any level of the hierarchy to a box at any other level so long as no cycle is created in the boxgraph.

The concept of *consistency* is fundamental to STL. Consistency is used for program control which emulates an IF-THEN construct without the introduction of specific control flow ideas. The data flow rule that allows this is 'data flow may not continue through an inconsistent box.' A box in a boxgraph becomes inconsistent when there is a conflict of some type among the data flowing into the box. Figure 2.2.5.1 shows a simple boxgraph both before and after execution. Boxes which are inconsistent are shown *shaded.* Boxes can be *open* or *closed.* Closed boxes, framed with solid lines, contain inconsistency; open boxes, framed with dashed lines, allow inconsistency to propagate out to the next larger containing box. Figure 2.2.5.2 shows how inconsistency affects data flow.

Iteration is available in STL via a construct called the *iteration box.* An iteration box stops iterating when it becomes inconsistent. Figure 2.2.5.3 shows a simple use of an iteration box. There are three forms of communication that can be used to send data to the internal section of an iteration box, sequential, parallel, and global. Sequential iteration is represented by pairs of triangle shaped *ports* on the frame of a box. Data values passed through sequential ports are sent back to the input of the box for subsequent iterations. Their values may change throughout the course of iteration. For example, in Figure 2.2.5.3, the value which arrives at the topmost sequential port on the right side of the iteration box after the first iteration (in this case the value is 5) is returned as an input value to the topmost sequential port on the left side of the iteration box. Ports are matched using simple lexicographic ordering of ports. Parallel ports, represented in STL as a small striped rectangle on the edge of a box, are used to transfer collections of data values to the groups of boxes represented by one iteration box. Any value coming into an iteration box via an arrow without a port (sequential or parallel) is considered to be a global value, which does not change throughout the iteration.

STL also includes box types which are used for opening files and accessing fields of a record structure. Naming of box graphs corresponds to the procedure concept in textual programming languages and can be used to simplify complex box graphs and to implement recursion. Also included in STL are a set of simple operation boxes, such as the plus box, '+', which perform the expected operations (addition, subtraction, multiplication, division, etc.)

2.2.6 LabVIEW (1986)

Laboratory Virtual Instrument Engineering Workbench (LabVIEW) [40] is a commercial visual programming environment designed for use by engineers and scientists with little or no traditional programming experience. The basic component of any program in the LabVIEW environment is the *virtual instrument* which consists of a front panel and a block diagram. As the name indicates, a virtual instrument is intended to imitate the behavior of a laboratory instrument, an oscilloscope for example. Figure 2.2.6.1 shows a virtual instrument with the front panel on the left and block diagram on the right. The block diagram is the source code, written in LabVIEW's visual programming language called G, which describes the virtual instrument's behavior.

G is a data flow based language with a special set of control flow structures added. The iconic nodes in G's block diagrams represent either predefined functions or user defined virtual instruments. The virtual instrument is G's procedural abstraction mechanism. The predefined functions are numerous and include arithmetic operators, comparative operators, trigonometric functions, string manipulation operators, statistical routines, matrix operators, curve fitting procedures, and signal processing routines. The added control flow structures include a sequence structure which forces its contents to be executed in a specified order, a FOR loop, and a REPEAT-UNTIL loop (mistakenly called a WHILE loop). Figure 2.2.6.2 shows a virtual instrument which uses a FOR loop construct in its block diagram. The directed edges between iconic nodes in G are called *wires* and their graphic rendering indicates the type of data which is flowing over them.

2.2.7 Forms (1987)

As one would expect from its name, Forms is a forms-based visual programming language. As was stated earlier, forms-based languages are an attempt to expand upon those aspects of spreadsheets that have made them a popular tool for "non-programmers" to use in solving problems on computers. One of those aspects is the visual layout of a spreadsheet as a matrix of cells. Another is the way in which cells can be specified when entering formulas by pointing a cursor at a cell or dragging the cursor across a group of cells instead of specifying the cells by name. Many of these properties can be summarized into the following statement. For a specific class of table-oriented computational problems, the spreadsheet solution "parallels the process we might use if we were to approach the problem with only pen and paper." [2] Forms-based visual programming languages attempt to maintain this property while expanding the class of problems that can be solved.

Some of the additions to typical spreadsheet concepts included in Forms are the following. The *sheet* concept is changed to a *form* which can contain objects such as cell matrices. Cells can contain numeric data, string data, or a graphic image. Cell matrices can be unbounded, that is their dimensions need not be specified until actual problem evaluation. Subforms are similar to forms except that they can inherit values from their parent form. This is similar to parameter passing, and subforms are the abstraction mechanism for creating subprograms or procedures. Subforms can be dynamically created during an evaluation; this allows procedures created with subforms to be recursive.

Figure 2.2.7.1 shows a subform which computes the binomial coefficients for an order N-1 equation where N is to be determined when the subform is invoked. This illustrates an unbounded cell matrix and the notation used to specify each cell's value based on its neighboring cells. The R stands for Row; C stands for column. Therefore, '=R[-1]C + RC[-1]' indicates the value of the cell one row up from the current cell in the same column should be added to the value one column to the left of the current cell in the same row to obtain the value for the current cell. The value of the single cell object N is to be inherited from the parent form. The simplicity and power of this notation for solving this type of problem is fairly obvious in the illustration.

2.2.8 Fabrick (1988)

Fabrick [22,29] is a visual programming environment intended to make programming a more natural process which is accessible to casual and novice programmers. It was developed in a commercial research environment at Apple

Computer. Programs are represented by data flow graphs of connected components. The components correspond to functions and are represented by icons. Fabrick extends the normal use of data flow to include bi-directional flow. Each component in Fabrick has a set of connection points, or pins, to which flow lines can be connected. Triangles represent the pins and indicate the direction of flow; bi-directional pins are represented by diamonds.

The pin metaphor is part of the overall electronics shop metaphor used in Fabrick. Programmers build new programs, or components, out of already defined components. A relatively large set of predefined components are included in the Fabrick environment. These include components which perform arithmetic, perform string and graphic manipulation, and generate common graphical elements (rectangles, ovals, lines, etc.) The system is set up to allow the addition of other kits containing special purpose components. Figure 2.2.8.1 shows the Fabrick Parts Bin (top) and a Construction Window (bottom). The parts bin is used to store previously created components. The programmer simply drags parts from the bin into the construction area and connects components by using a mouse to draw lines between pins.

The component system is Fabrick's main abstraction mechanism. Figure 2.2.8.2 shows a completed analog clock component created using previously built components. Once the clock component is built, it can be represented by just the part of the diagram that looks like a clock face. This icon can then be added to the bin of available components and used to create other components.

There are three other aspects of Fabrick which are particularly interesting. First, it contains a simple gesture recognition system to increase the number of commands that can be associated with a mouse button. After positioning the cursor over an object and pressing the mouse button, moving the mouse in a specific direction indicates the operation to be performed. For example a pin can be connected to by clicking on the pin and moving the cursor in one direction. That same pin can be moved around the perimeter of its component by clicking on the pin and moving the cursor in a different direction. Second, unlike many programming languages and systems, Fabrick is fully live. This means that whenever a connection, value, or component is changed, the rest of the current graph is immediately updated to reflect the change. Lastly, Fabrick contains a group of components called iterators which are used for computations which are to be repeated. These iterators function in a manner which is similar to the iterators of Show and Tell described above.

2.2.9 Cantata (1990)

Cantata is a data flow based visual programming language which is part of an integrated software development environment call Khoros [33,43]. The system's designers contend that most data flow based visual programming languages are too limited in application and that they need to be made more general in order to be widely accepted and used. Cantata uses a directed graph of boxes, called *glyphs,* as its standard notation. However, Cantata attempts to provide the appropriate compromise between visual and textual programming. Thus, operators, those fundamental procedures which are represented as glyphs, are written in traditional programming languages such as C or FORTRAN. This allows numerous pre-written libraries of procedures to be quickly incorporated into the environment. This also reflects the system designer's point of view that some things are better represented in textual rather than graphical notation.

Figure 2.2.9.1 shows the Cantata workspace with the source code of an image processing application. The "count_loop" in Figure 2.2.9.1 illustrates one of Cantata's control flow constructs, counted iteration. A WHILE loop and an IF-THEN-ELSE construct are also included. Procedural abstraction is provided by Cantata; new glyphs can be made by creating a flow graph consisting of existing glyphs.

The Khoros/Cantata environment also takes on some features of a forms-based language. Glyphs are partitioned into groups of similar functions which are accessible through a forms interface. Figure 2.2.9.2 shows a subform and its corresponding flow graph implementation. The "Standard Control Structures" are all accessed through the same form. The subform shown is for the IF-THEN-ELSE control structure.

2.2.10 Hyperflow (1992)

Hyperflow [23, 24] is a visual programming language specifically designed to work in a pen computer environment. It is an extension of the Show and Tell Language discussed above, but it is also a user interface framework. In that sense, it is also an extension of various window systems such as X windows [36].

The basic building block of Hyperflow is the visually interactive process, or *vip*. A vip is a concurrent process with a user interface. This user interface is typically represented by a box or group of boxes on screen, but can be represented by various on-screen renderings. A vip's user interface corresponds to a window in a traditional windowing system and can respond to user input in a similar manner. Along with the user interface, each vip has a processing part which contains, among other things, the specification of how the vip is to respond to commands, whether they be from user input or from messages from other vips. It is this message passing and receiving capability of vips which makes the group of vips a *communicative organization* capable of distributed computation. Vips are connected by arrows indicating communication paths. A vip's body, its processing part, can be encoded in the two dimensional syntax of Hyperflow or in a textual programming language such as C, Pascal, or even assembly language.

Figure 2.2.10.1 shows the specification of a vip which is used to calculate the Fibonacci sequence. Boxes which are connected by lines, not arrows, are all part of the same vip. This particular vip has two commands which it can accept, *init* and *next*. Init is encoded in a textual language, while next is encoded in the two-dimensional Hyperflow syntax. After init is requested, subsequent invocations of the next command will supply the successive integers of the Fibonacci sequence. Figure 2.2.10.2 shows how that vip is used to compute the 10th Fibonacci number.

It is worth noting that every functional module in a Hyperflow system, including such low level items as device drivers, has a corresponding vip and thus has an interface part and a processing part. This approach reflects the position that "the user interface is as ubiquitous as the computation." [23] One of the goals of the Hyperflow language is that it be able to function as a common language at the various levels of programming from user interface through system programming.

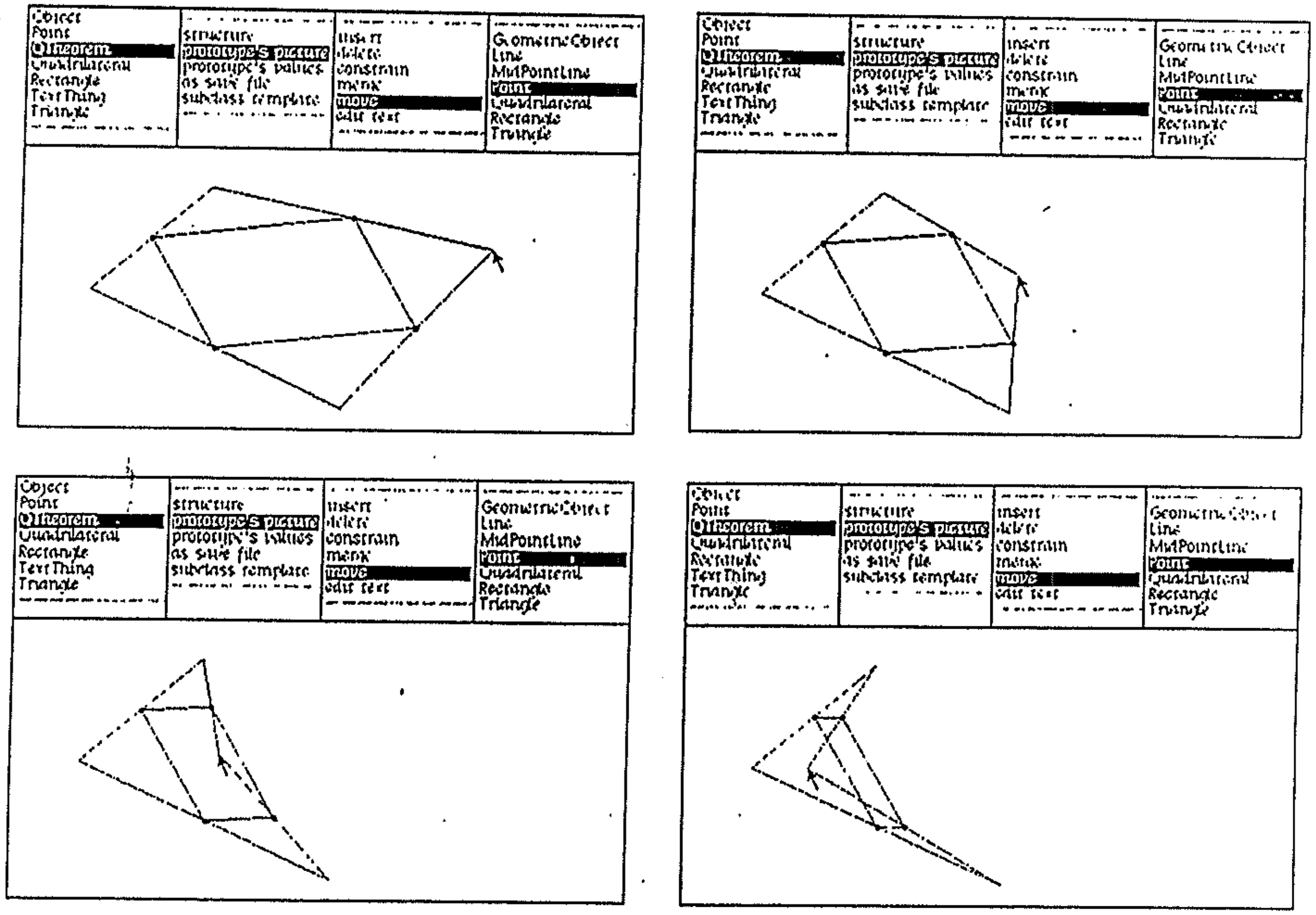


Figure 2.2.1.1: Moving a Vertex of a Quadrilateral Object

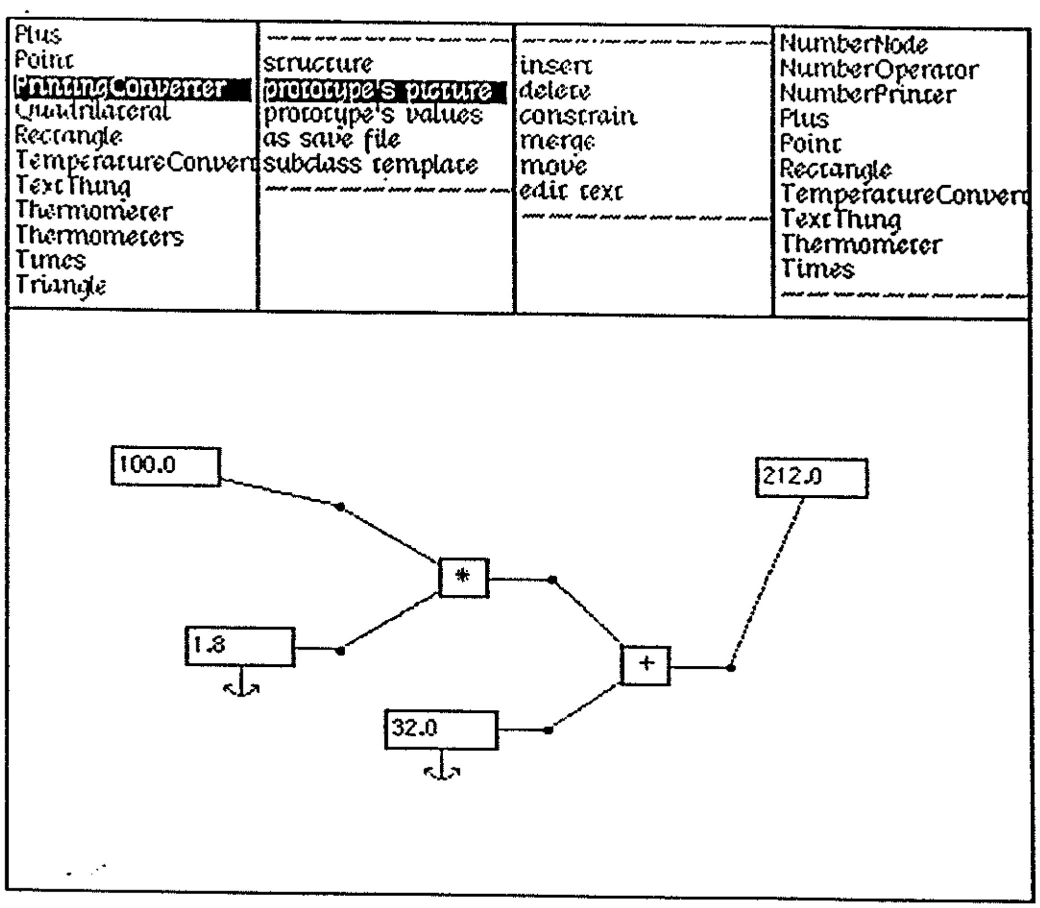
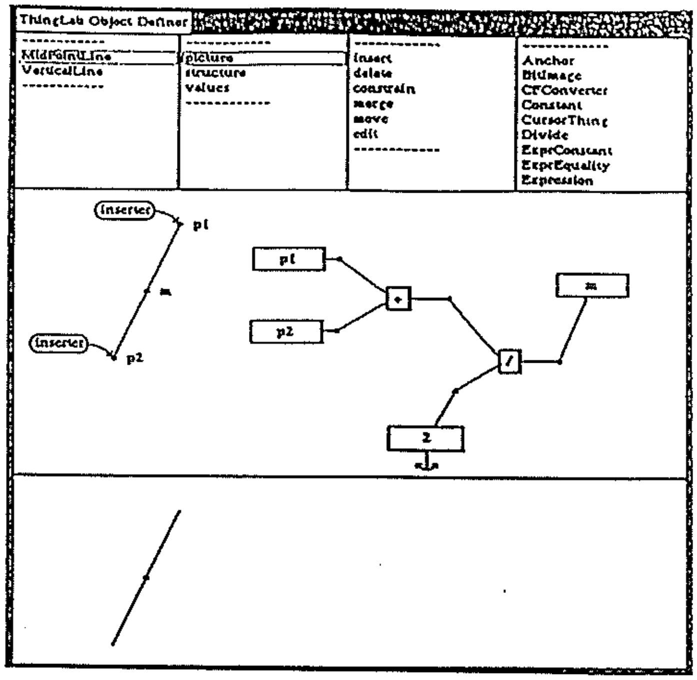


Figure 2.2.1.2: A Celsius-to-Fahrenheit Converter



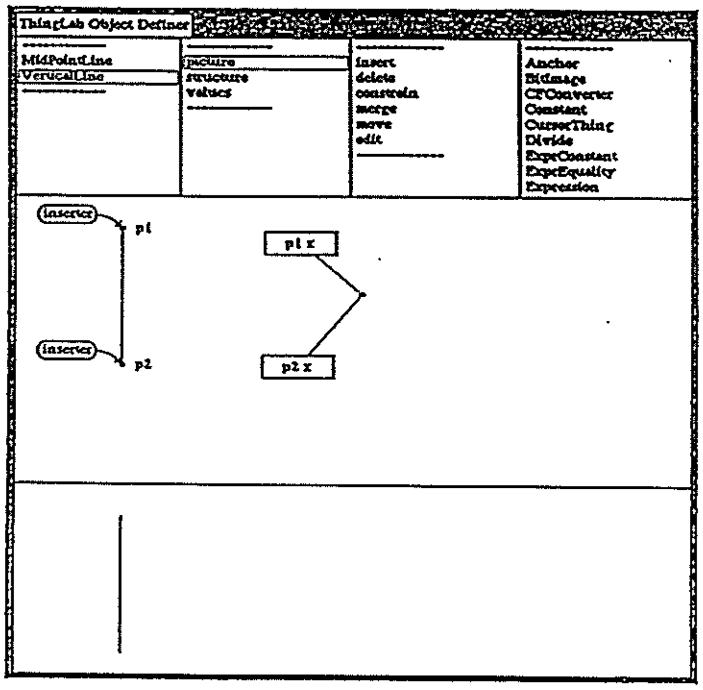
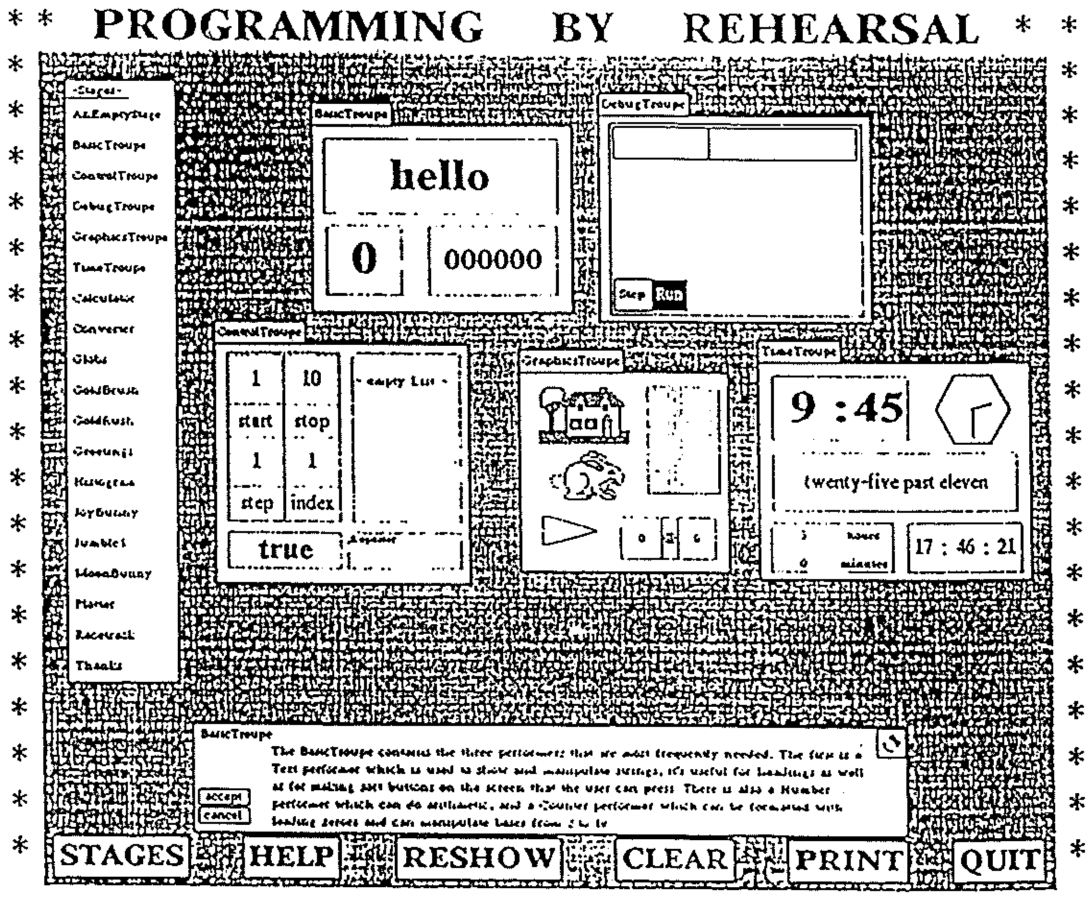


Figure 2.2.1.3: Graphically Defining the New Objects MidPointLine and VerticalLine



BasicTroupe

The BasicTroupe contains the three performers that are most frequently needed. The first is a Text performer which is used to show and manipulate strings, it's useful for headings as well as for making text buttons on the screen that the user can press. There is also a Number performer which can do arithmetic, and a Counter performer which can be formatted with leading zeroes and can manipulate bases from 2 to 16

Figure 2.2.2.1: Troupes of Performers in Rehearsal World

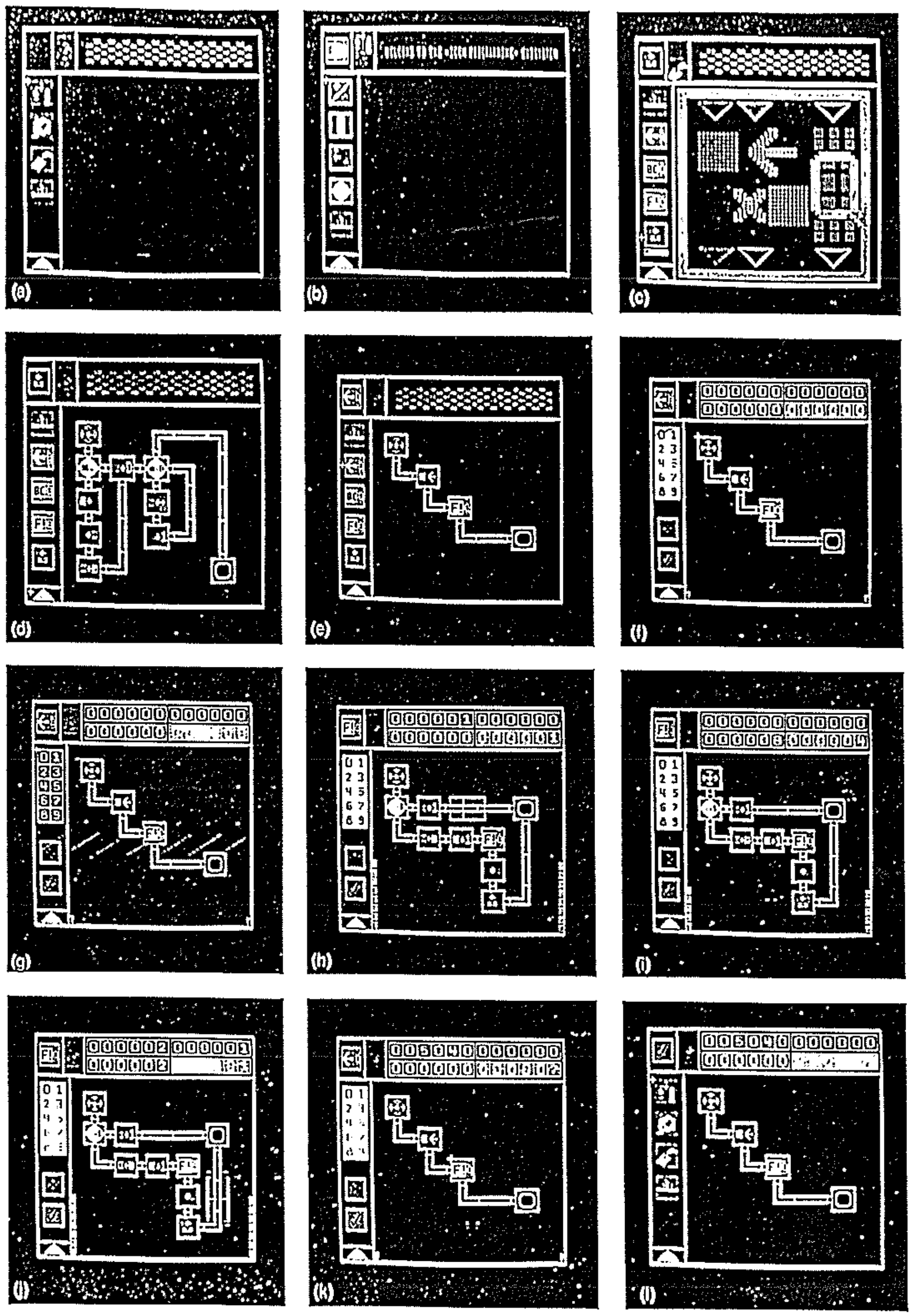
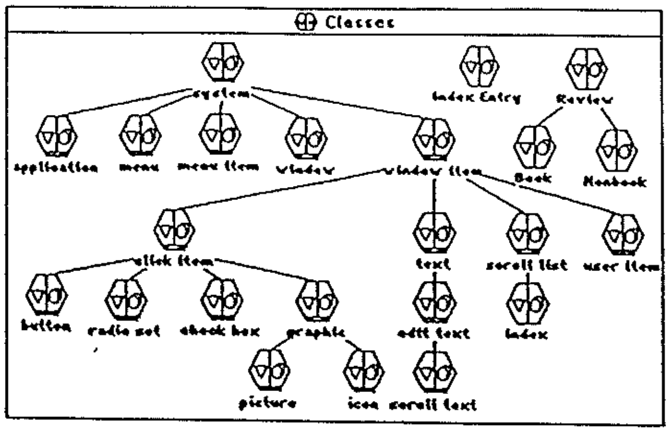
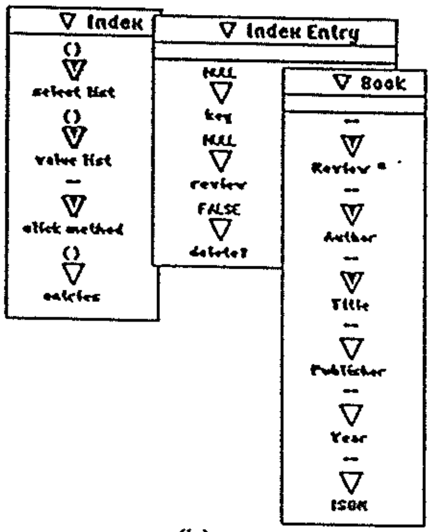


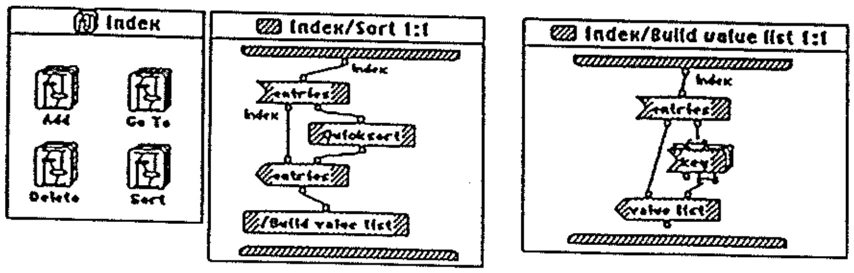
Figure 2.2.3.1: Screen Images from Pict



(a)

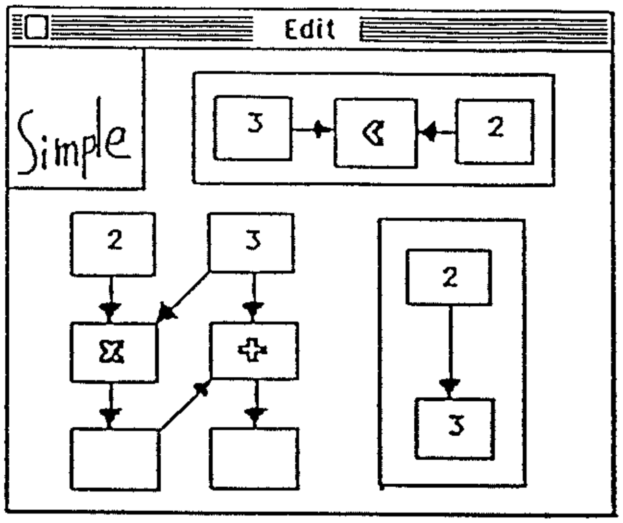


(b)

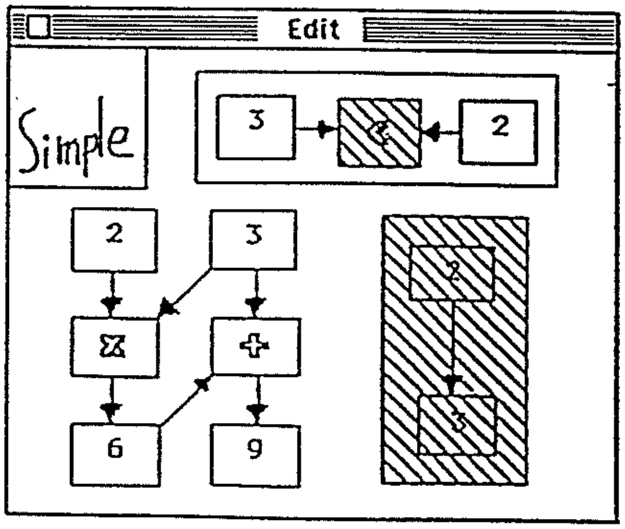


(c)

Figure 2.2.4.1: Prograph Classes



(a) Simple Box-Graph before Execution



(b) Simple Box-Graph after Execution

Figure 2.2.5.1: A Simple BoxGraph

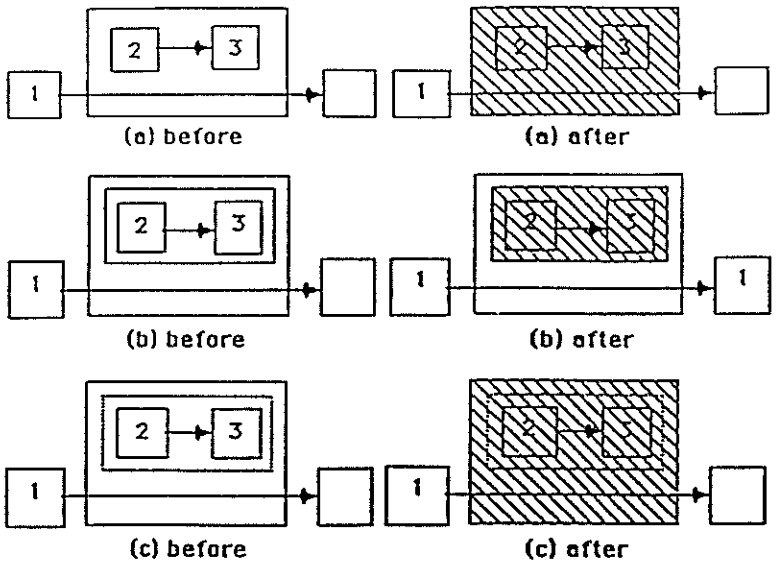


Figure 2.2.5.2: Data Flow and Inconsistency

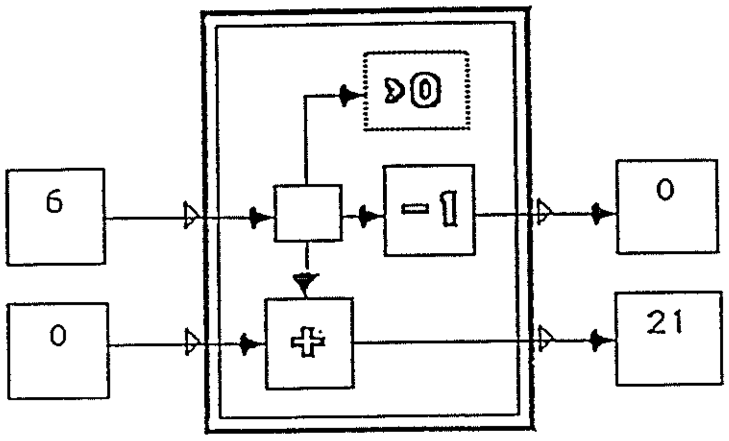


Figure 2.2.5.3: A Simple Iteration Box

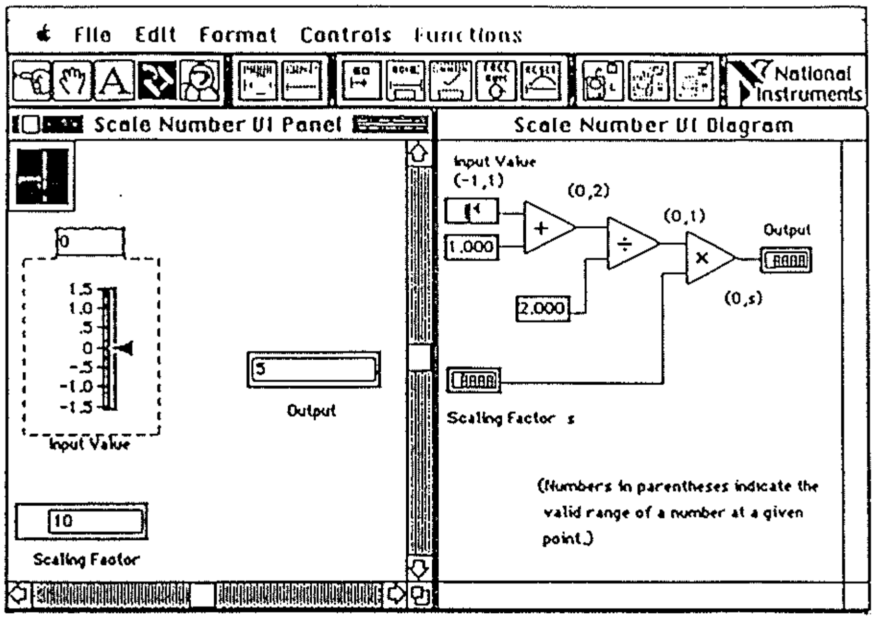


Figure 2.2.6.1: A LabVIEW Virtual Instrument

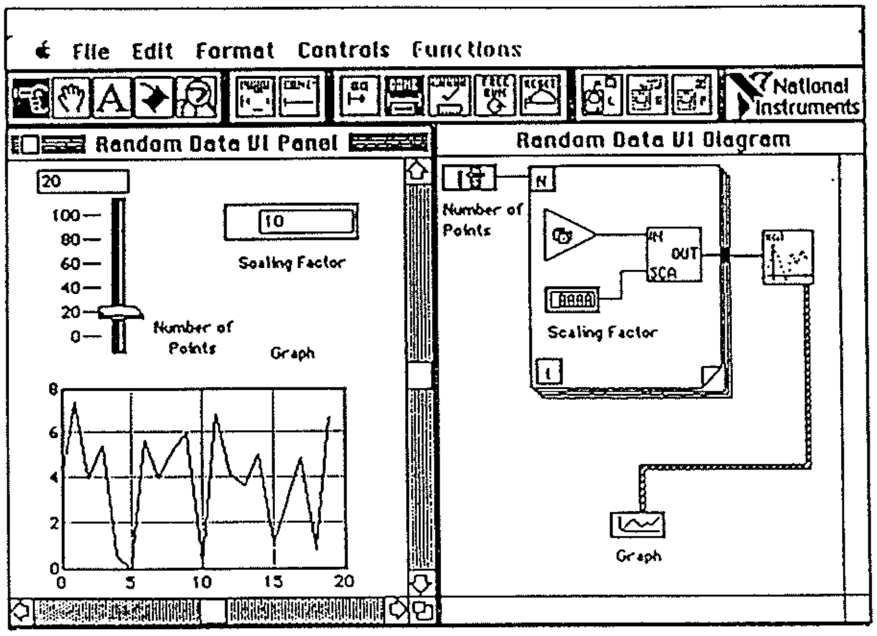


Figure 2.2.6.2: A Virtual Instrument with a FOR Loop

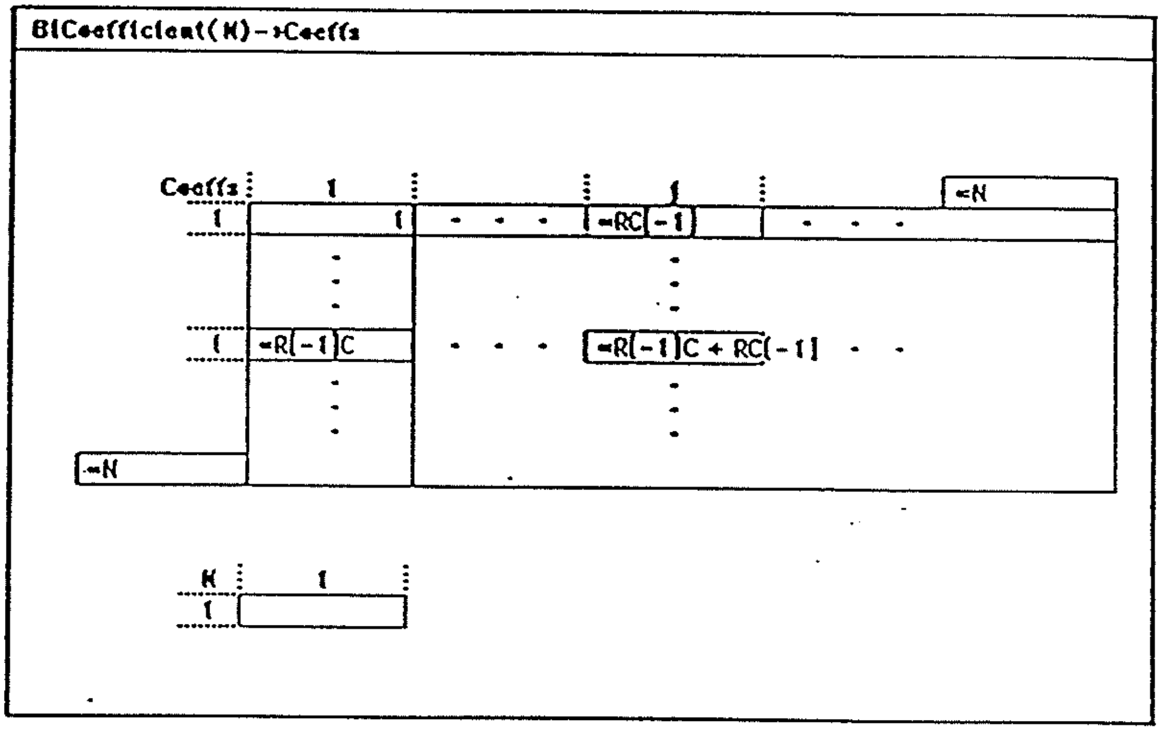
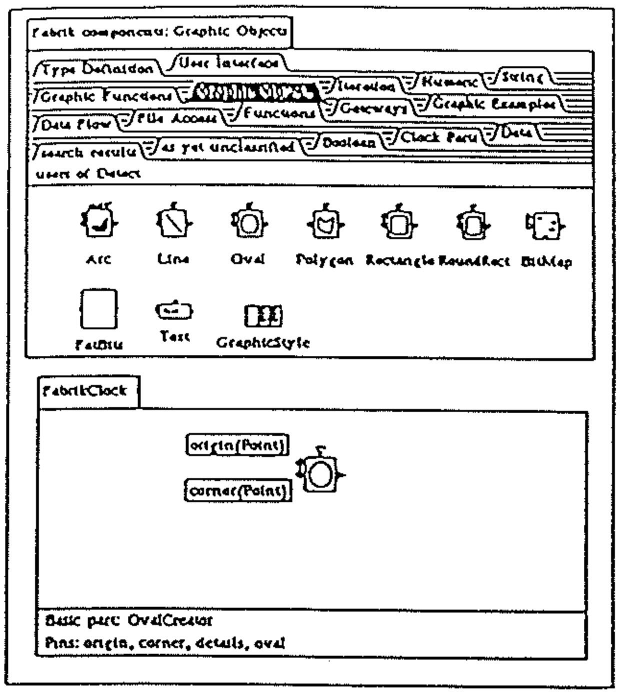
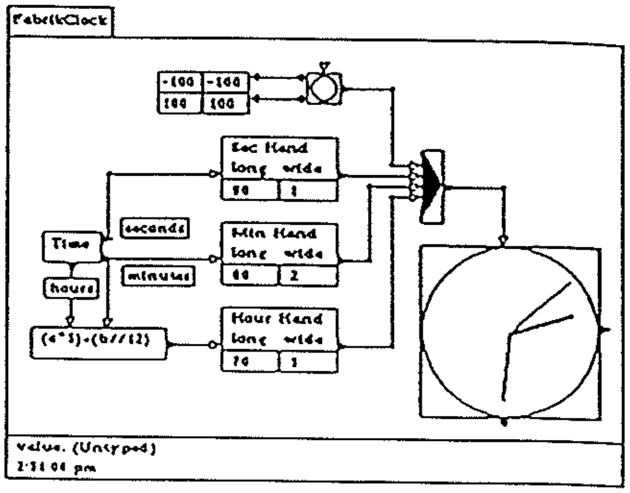


Figure 2.2.7.1: The Binomial Coefficients Subform



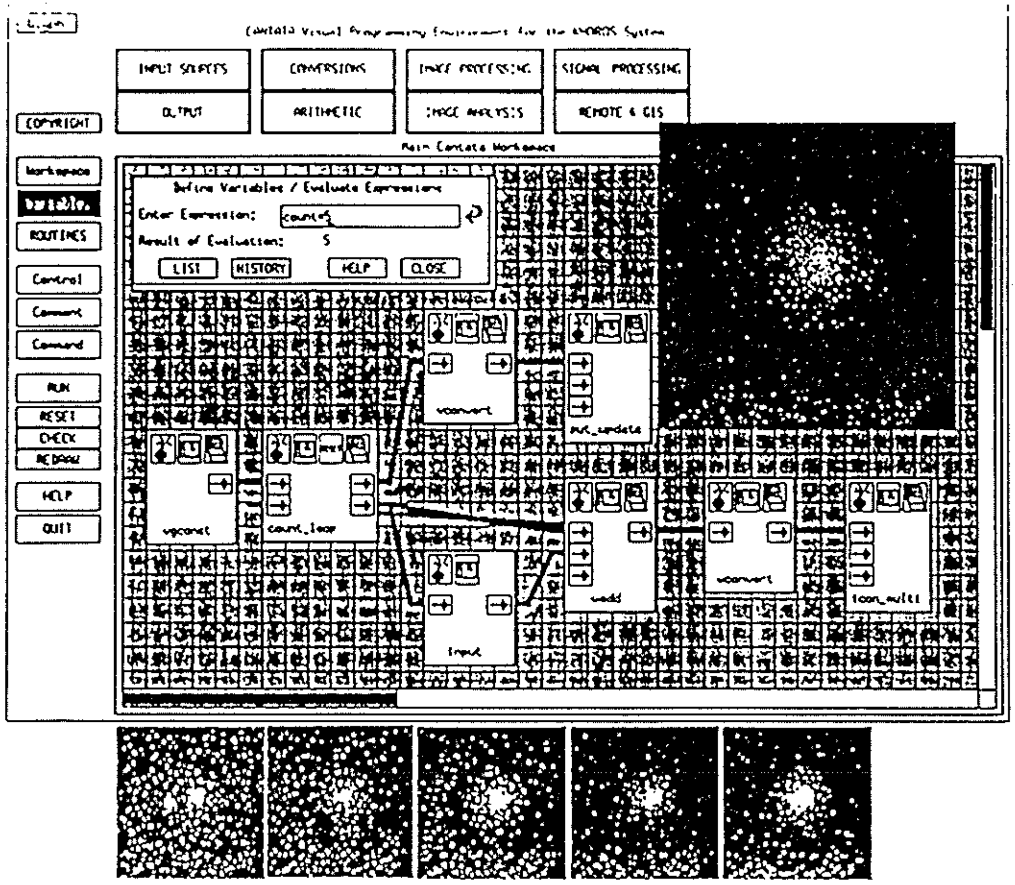
Basic part: OvalCreator
Pins: origin, corner, details, oval

Figure 2.2.8.1: A Parts Bin and Construction Window



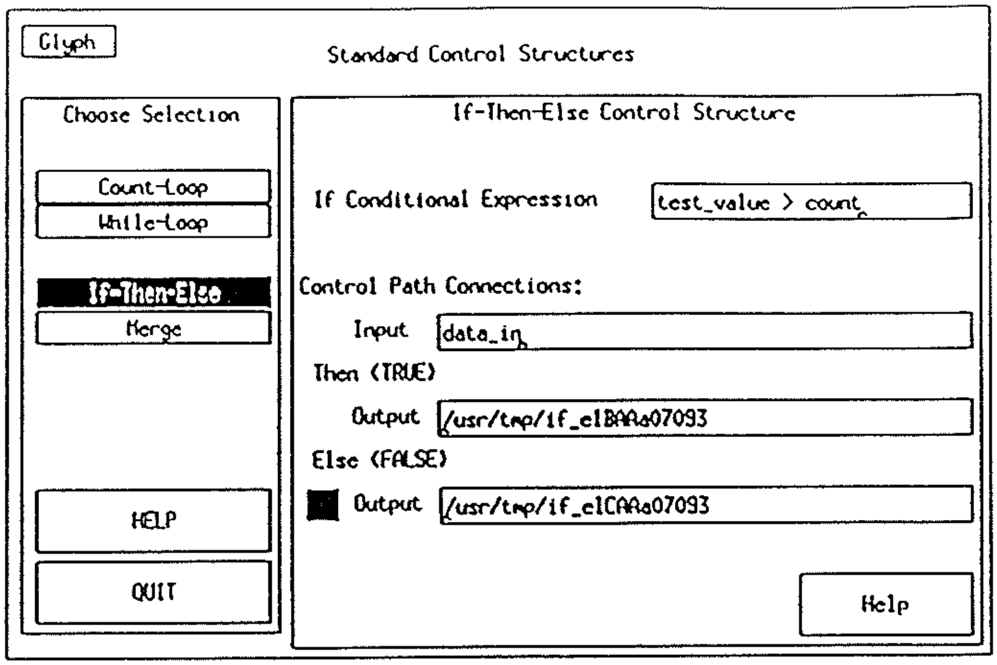
Value. (Untyped)
2:34:04 pm

Figure 2.2.8.2: FabrickClock



Enter Expression: count=5
Result of Evaluation: 5

Figure 2.2.9.1: Cantata Workspace



Glyph

Standard Control Structures

Choose Selection

Count-Loop

While-Loop

If-Then-Else

Merge

HELP

QUIT

If-Then-Else Control Structure

If Conditional Expression: test_value > count

Control Path Connections:

Input: data_in

Then (TRUE)

Output: /usr/tmp/if_elBAAa07093

Else (FALSE)

Output: /usr/tmp/if_elCAAa07093

Help

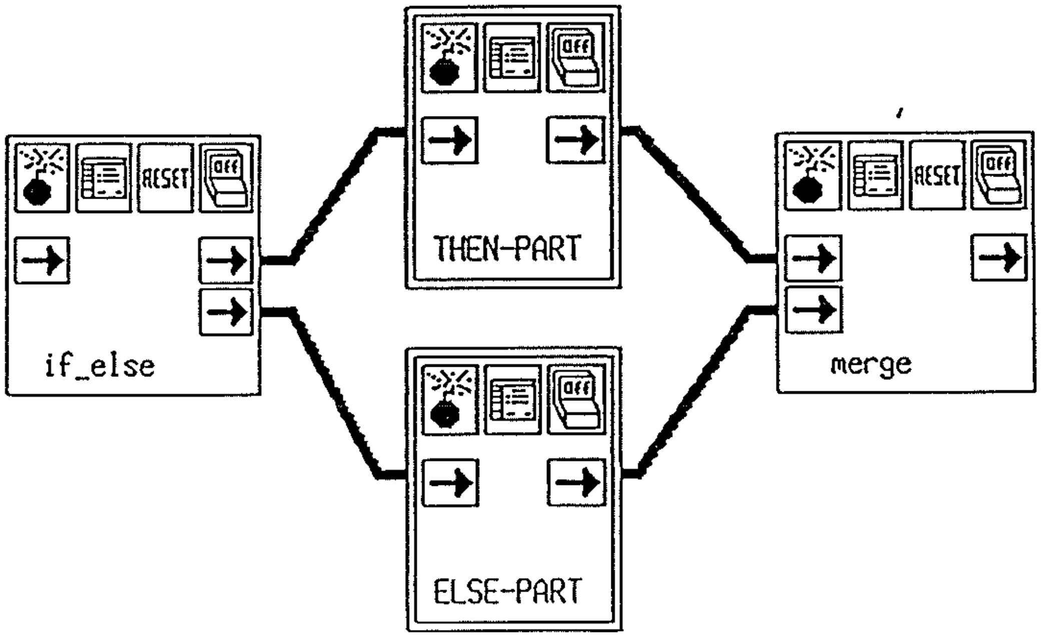


Figure 2.2.9.2: A Subform and Corresponding Flow Graph

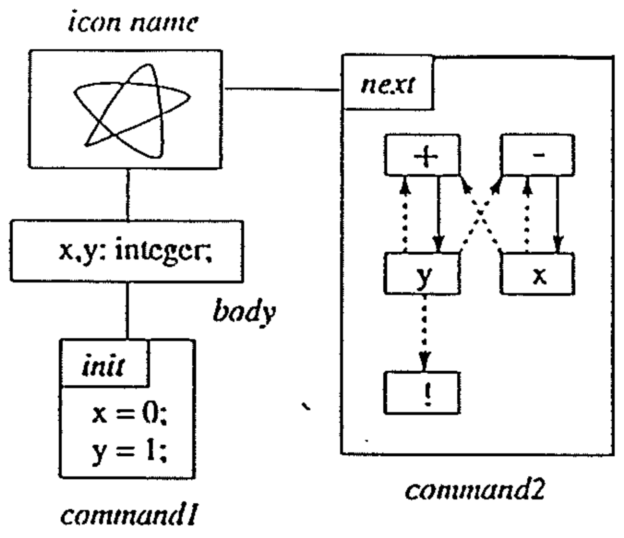


Figure 2.2.10.1: A Fibonacci vip

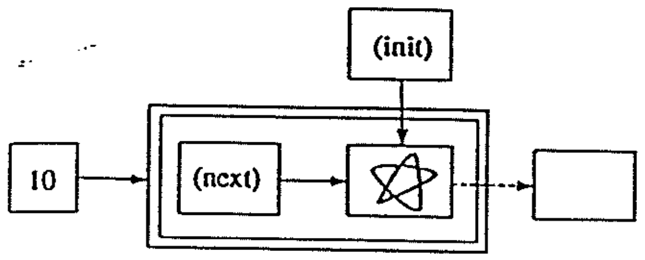


Figure 2.2.10.2: Use of the Fibonacci vip

2.3 Motivation

The examples presented in the preceding section provide a brief overview of the visual programming languages which have been designed. They are presented chronologically in order to highlight a trend. The goals of visual programming languages are becoming more and more ambitious. Early visual programming languages were designed for a small class of users and a small class of problems. Newer visual programming languages are building on their predecessors to become much more general purpose.

This trend is to be expected, because, where programming languages are concerned, there is a large gap between the way human beings conceive of solutions to problems and the way we must program those solutions for computers. Visual programming promises to narrow that gap. To the extent that it does narrow that gap, visual programming is desirable at all levels and for all users. However, as Glinert [13] states, this promise will not be fulfilled until the truly fundamental open problems of visual programming are solved. He goes on to state

> One of the major obstacles preventing full realization of the visual approach's potential is the present dearth of formal underpinnings for the field. Even seemingly simple things such as good notations analogous to the BNF which has traditionally been used to precisely and unambiguously describe textual programming languages seem hard to come by in the visual and iconic cases.

We believe that, in order to support the trend towards more general purpose systems, the complementary path towards the foundations of the field must be followed. We are motivated by the desire to help provide some of the formal underpinnings for the field. In particular to describe a visual language which is analogous to the Turing machine. Those properties of a Turing machine which we would like our visual language to have are *simplicity* and *computational completeness*. How we produce and measure simplicity and completeness is the subject of the next section.

3. Methods and Metrics

Parsimony is, of course, a subjective concept. But we can try to use methods which create a simple language and try to measure the simplicity of the language. The visual languages presented in the previous section were identified by their semantic base, their syntactic base, and the additional constructs added to then language. We have attempted to choose the simplest syntactic and semantic bases for our visual language and have attempted to add as few additional constructs as possible.

The semantic base chosen is data flow. In order to appreciate the simplicity of this model, we note that such systems are built around a simple *data availability firing rule*: An operator or operation executes when and only when its necessary input values are available and produces output which is then sent to other operators which need these values. An operation in the pure data flow model has no other side effects; that is, no other values are changed. In contrast to the von Neumann model of computation, no addresses or program counters are part of the model. Also, there is an equivalence between the data dependencies and the operation scheduling constraints. That is, there are no operation sequencing constraints other than the ones imposed by the data dependencies. Also, there is no notion of a "single locus of control." Any two operations may be executed concurrently. [1]

With the semantic base chosen, the choice of a syntactic base becomes relatively easy. The data flow model is easily and naturally represented by a directed graph whose nodes represent operations and whose arcs represent data dependencies between operations. This syntax is very simple, node-arc-node.

We would like to add as few other constructs to the model as possible. How well we meet this goal can be measured by three metrics. The first metric is the number of visual components of the syntax. Obviously, we need a node representation, in this case a box, and a dependency representation, in this case an arrow. We would like to use only these two visual components. The second metric is the number of functional components. That is, the number of "types" of nodes. In many data flow based visual programming languages there are a large number of node types, sequencing nodes, IF-THEN-ELSE nodes, WHILE loop nodes, and iteration nodes for example. We would like to keep this number to a minimum. Having a small number of node types means minimizing the number of decisions about node execution that need to be made by the program execution engine. Thirdly, we would like to maintain the smallest possible number of types of relationships among nodes. We are required to have the data dependency relationship, but we would like to add no other relationships. In particular, the position of nodes should have no influence on their execution.

The constructed computation model is not only to be simple, but also computationally complete. The measure of computational completeness is the Turing machine and the ability to solve all computable functions as defined by the Turing machine. We will show our computation model to be complete in that sense.

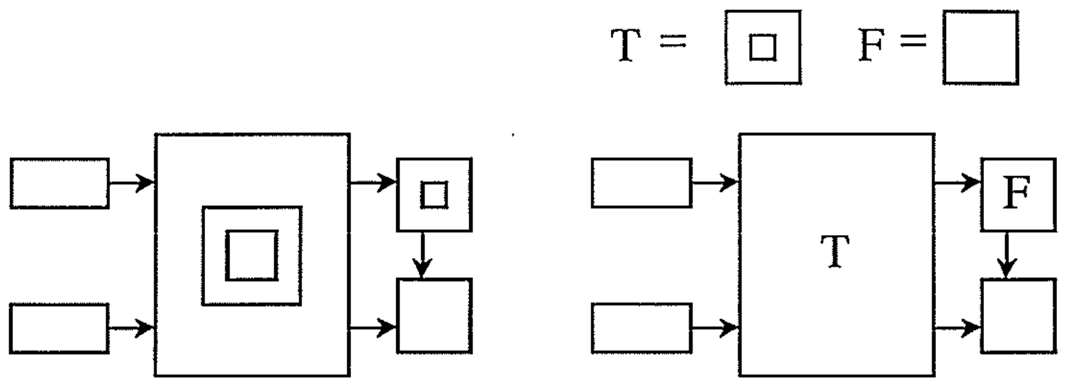
4. The Boxgraph Computation Model

In this section we present our computation model in two parts. First, the basic boxgraph model, which is equivalent to Propositional Logic, is presented. Then the basic boxgraph is extended to make the Recursive Boxgraph model which is computationally complete.

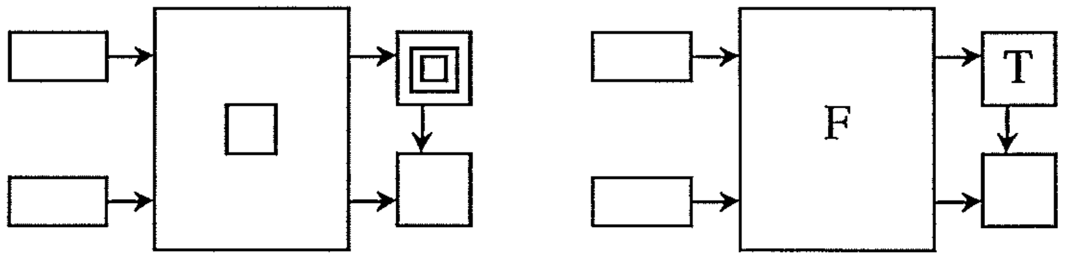
4.1 Basic Boxgraph

The Boxgraph model of computation [25] has data flow as its semantic base and the directed graph as its syntactic base. The model consists of a set of diagrams in the two-dimensional plane and a set of transformation rules. A *boxgraph* is an acyclic directed graph of hierarchically nested boxes. We leave further discussion of the use of the term acyclic in this context for later. The visual components of the model are boxes and arrows. No text is needed. Boxes are used not only to represent operational nodes of the graph but also to represent data.

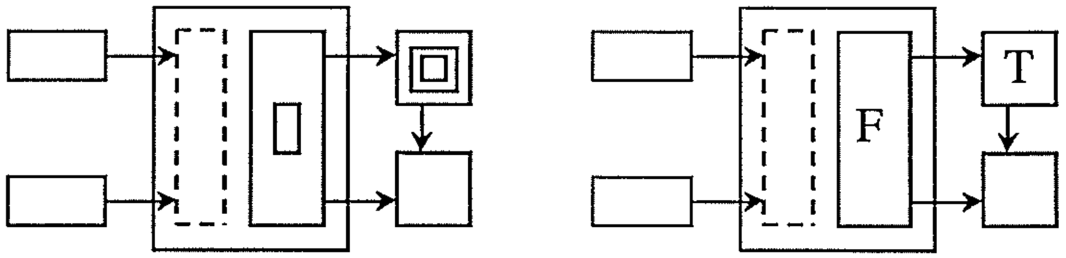
Figures 4.1.1 and 4.1.2 show examples of boxgraphs. Figure 4.1.1 shows the logical functions AND, OR, and XOR, where the logical values, True (T) and False (F), are represented by a double box without arrows and a single box without arrows, respectively. It is important to note that "T" and "F" are not part of the model. They are only added for convenience of representation. In each of these functions, the result of the operation will be the value that flows into the lower right-hand box of the graph. Figure 4.1.2(a) shows the boxgraphs before and after the computation of F=AND(F,T), and Figure 4.1.2(b) shows the same for T=AND(T,T). These executions illustrate the concept of *inconsistency*. Inconsistency in the boxgraph model is very similar, but not identical, to inconsistency in the Show and Tell Language.



(a) AND Function

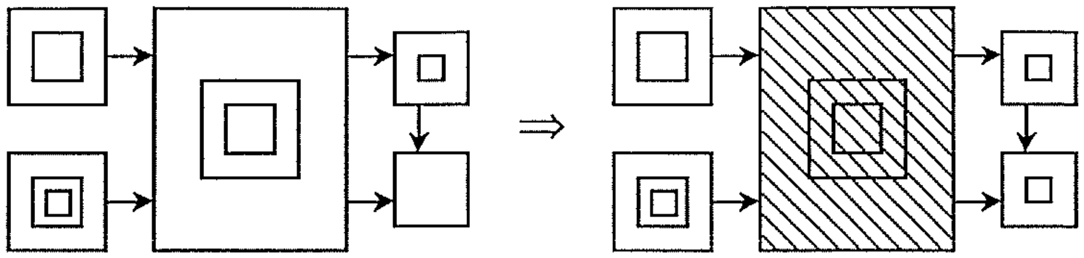


(b) OR Function

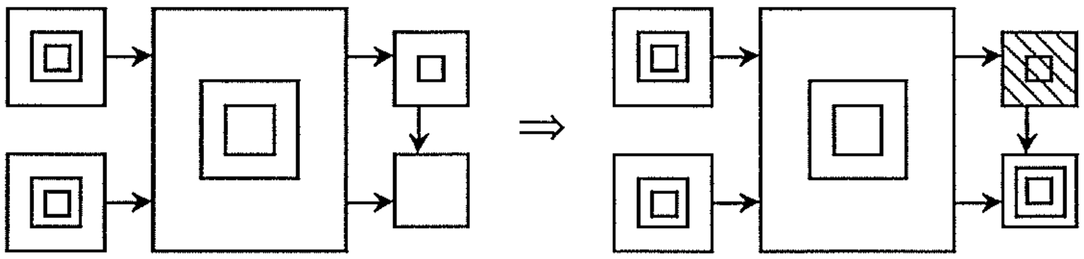


(c) XOR Function

Figure 4.1.1: Logical Functions in Boxgraph



(a) Computation of F=AND(F,T)



(b) Computation of T=AND(T,T)

Figure 4.1.2: Computations of AND

During the computation, a box may become inconsistent when two different values try to occupy the same box causing a conflict. This is shown by shading or cross hatching the inconsistent box. When a box becomes inconsistent, all the arrows incident with the box become non-existent for the remainder of the computation. This includes arrows which arrive at the box, those that leave from the box, and those which pass through the interior of the box. A box can be either *open* or *closed* in terms of limiting the scope of inconsistency propagation. An open box, which is framed by dashed lines as in Figure 4.1.1(c), allows inconsistency to flow out to its surrounding environment. When an open box becomes inconsistent, the smallest box containing it also becomes inconsistent. A closed box, which is framed by solid lines, does not propagate inconsistency to its environment. The concept of inconsistency will be defined in more exact terms later.

A box partitions the two-dimensional space into two parts, the interior and the exterior of the box. The interior defines the functionality of the box and the exterior defines the usage of the box. Using the concepts of interior and exterior, we can now give a more exact definition of a cycle in a boxgraph and thus clarify the use of the term acyclic. Let X and Y be two distinct nodes (boxes). Let X be exterior to Y. By definition then Y is also exterior to X. A *cycle* is formed between X and Y if there is a path of arrows which starts at X or the interior of X and arrives at Y or the interior of Y, **and** there is another path which starts at Y or the interior of Y and arrives at X or the interior of X. Such a cycle is a problem for our model because it creates a cycle of data dependency. If such a cycle exists, then Y depends on data produced by X and X depends on data produced by Y. This implies that X must fire and produce output before Y can fire, and Y must fire and produce output before X can fire. The term *acyclic,* when applied to boxgraphs, means that no such cycles exist.

The location, size, and shape of a box are insignificant. Indeed a box can be replaced by any simple closed curve. Similarly, the size and shape of an arrow are insignificant. The location of an arrow, however, is significant in terms of which boxes it intersects.

No two boxes may intersect with each other, but an arrow may intersect with other arrows and with boxes. An arrow can cross box boundaries to connect any two boxes. Provided, of course, that no cycle, as described above, is created by the arrow. The graph in figure 4.1.3(a) is not a boxgraph because a cycle is formed between boxes **A** and **B** by the arrows α and β.

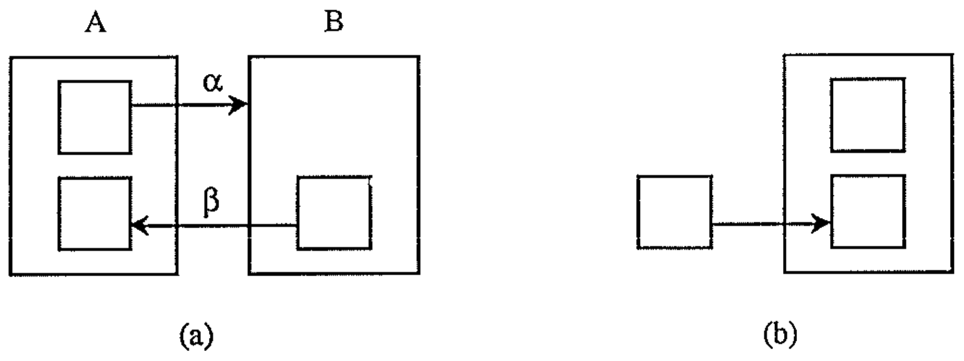


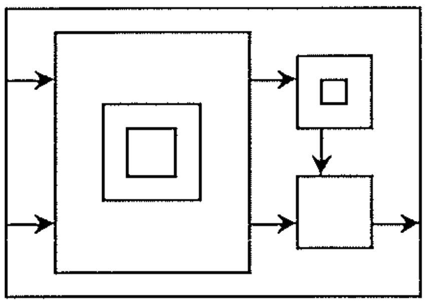
Figure 4.1.3: Non-Boxgraphs

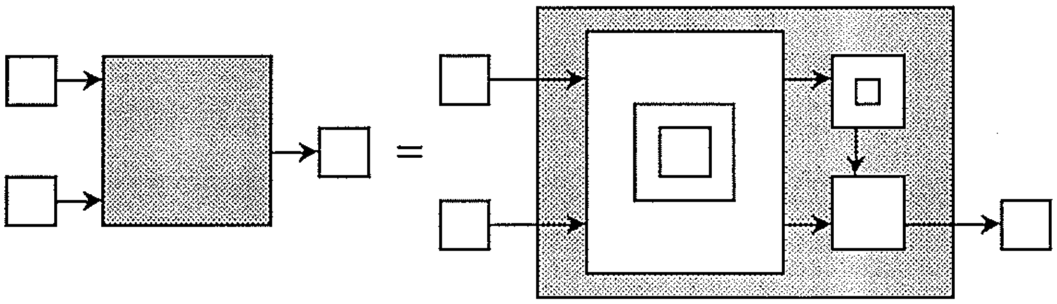
A boxgraph containing no arrow is called *trivial* and represents a *value* such as T or F in Figure 4.1.1. A box may contain either nothing, a trivial boxgraph, or a non-trivial boxgraph, but cannot contain both a trivial boxgraph and a non-trivial boxgraph. Figure 4.1.3(b) illustrates a non-boxgraph which violates this rule. A box containing a non-trivial boxgraph, such as the box shown in Figure 4.1.4(a), is called an *operation* box. An empty box or a box containing a trivial box graph is called a *memory* box. In our intended semantics, an operation box receives input values from its exterior and returns output values to its exterior. Arrows which arrive at a box boundary, whether from the interior or exterior, are referred to as *in-arrows* with respect to the box. Arrows which start at a box boundary are referred to as *out-arrows* with respect to the box. Establishing the association between the in-arrows and out-arrows of a procedure box is analogous to parameter binding in traditional programming. The boxgraph model uses the following *positional binding rule:*

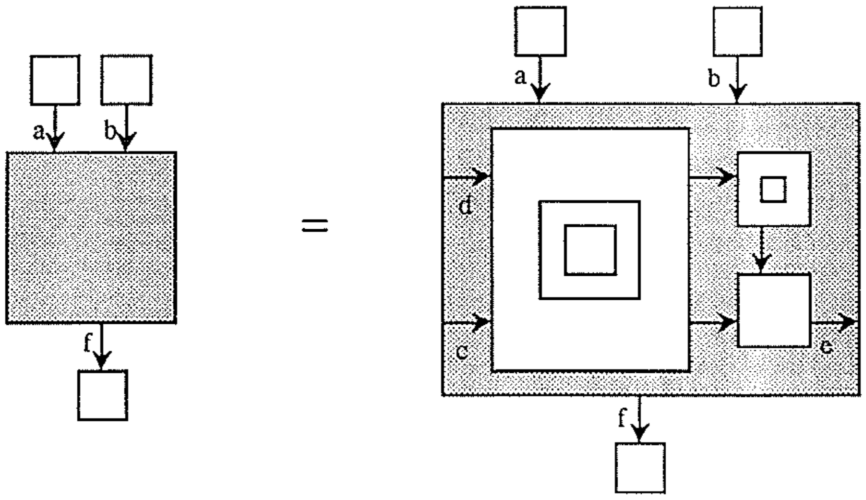
> Rank all the in-arrows from the exterior by lexicographic ordering of the (x,y) coordinates of their intersection points with the box. Then do the same for all the out-arrows to the interior. Bind an in-arrow with an out-arrow of the same rank.

The same binding rule applies to in-arrows from the interior and out-arrows to the exterior. An arrow which crosses a box boundary can be seen as an abbreviation for two arrows, an in-arrow which arrives at the boundary and out-arrow which leaves the boundary from the same location. Figure 4.1.4(b) shows the use of an operation box. An example of the positional binding rule is shown in Figure 4.1.4(c). Arrows **a** and **b** are bound with arrows **c** and **d** respectively, and arrow **e** is bound with arrow **f**. An operation box defines an environment for the computations carried out by its interior boxes and controls propagation of inconsistency to its outer environment. In that sense it defines a two-dimensional block structure.

The informal semantics of the boxgraph model can be presented in an imperative (prescriptive) form as follows: A trivial boxgraph represents a *datum* or a *value*. A non-trivial boxgraph represents an *operation*. An arrow represents a data communication path, i.e. a data flow. A value *flows* from the starting box of an arrow to the ending box. The data transfer may be carried out anytime asynchronously. Since there are no cycles, once a box is filled by a value, the value stays there. A memory box becomes *inconsistent* if the incoming value is different from the value already existing in the box (if such a value exists), i.e. when a conflict occurs in the box, it becomes inconsistent. When a box becomes inconsistent, all arrows intersecting with the box become ineffective, i.e., no data may flow on those arrows. If an open memory box becomes inconsistent, the smallest operation box containing the open box also becomes inconsistent. If the operation box is also open, the smallest box containing the open operation box becomes inconsistent, and so on. A computation halts when all empty boxes are filled with values.


(a) Operation Box


(b) Using Operation Box - I



(c) Using Operation Box - II

Figure 4.1.4: AND Operation Box

We can make the above semantics more precise, by presenting them in a declarative (descriptive) form. In the declarative semantics, a boxgraph defines a set of logical constraints on the box contents. A boxgraph is consistent if the content of each box satisfies the constraints imposed by the neighboring box contents. In this form, the data flow semantics show a close relationship to the constraint based semantics of other visual languages. A computation is defined as the process of finding a set of values for filling in the empty boxes without violating the consistency of the boxgraph. An *elaboration* of a boxgraph is an assignment of values, including the null value ($\perp$), to all the arrows such that the constraints imposed by each box may be satisfied. Formally a boxgraph is *consistent* if there exists an elaboration, otherwise it is *inconsistent*. The concept of elaboration is similar to the concept of interpretation in mathematical logic, where a set of propositions are defined to be consistent if there exists a model (interpretation) for which all propositions are satisfied. As in logic, the consistency of a boxgraph depends on the existence of an elaboration and not on how the elaboration is constructed. Local constraints imposed by a box on the values of the incident arrows are defined as follows (see Figure 4.1.5):

Let $A = \{a_1, a_2, \ldots, a_m\}$ be the set of values on the in-arrows to a box,

and $B = \{b_1, b_2, \ldots, b_m\}$ be the set of values on the out-arrows from a box.

(1) When the box is a memory box with α ($\perp$, if it is empty) as its content:

(1.1) If $A \cup \{\alpha\}$ has no non-null value, then $B = \{\perp\}$.

(1.2) If $A \cup \{\alpha\}$ has exactly one non-null value a, then $B = \{a\}$.

(1.3) If $A \cup \{\alpha\}$ has more than one non-null value and the box is closed, then $B = \{\perp\}$.

Note that if $A \cup \{\alpha\}$ has more than one non-null value and the box is open, then there is no assignment of any value to B which preserves consistency.

(2) When the box is an operation box with β as its content:

(2.1) β bound with A and B is consistent.

(2.2) If the box is closed and β bound with A and B is inconsistent, then $B = \{\perp\}$.

Note that if the box is open and β bound with A and B is inconsistent, then there is no assignment of any value to B which preserves consistency.

If a boxgraph is consistent and an elaboration is found, then the computation of filling the empty boxes can be carried out by transferring the values on the in-arrows into the empty boxes. As an example, for computing T=XOR(F,T) and F=XOR(T,T), using the boxgraph of Figure 4.1.1(c), elaborations are given in Figure 4.1.6. The result of a computation is unique if the elaboration is unique.

In order to demonstrate the power of the boxgraph model presented so far, a binary full adder is constructed in Figure 4.1.7. With x, y, and c as a Boolean input, the Boolean output values, s and c', are computed by: $s = x \oplus y \oplus c$ and $c' = xy + yc + xc$.