

Visual Languages and Computing Survey: Data Flow Visual Programming Languages

DANIEL D. HILS

*Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W.
Springfield Avenue, Urbana, Illinois 61801, U.S.A.*

Received 20 November 1990 and accepted 25 May 1991

The data flow model is a popular model on which to base a visual programming language. This paper describes alternatives available to a designer of data flow languages, describes many of the languages, discusses some strengths of the languages, and discusses some unsolved problems in the design of data flow languages.

1. Introduction

DATA FLOW IS A POPULAR COMPUTATIONAL MODEL for visual programming languages. Data flow provides a view of computation which shows the data flowing from one filter function to another, being transformed as it goes. In addition, the data flow model easily accomodates the insertion of viewing monitors at various points to show the data to the user. Consequently, many recent visual programming languages are based on the data flow model.

This paper describes many of the data flow visual programming languages. The languages are grouped according to their application domain. For each language, pertinent aspects of its appearance, and the particular design alternatives it uses, are discussed. Next, some strengths of data flow visual programming languages are mentioned. Finally, unsolved problems in the design of such languages are discussed.

2. Methods for Classification of Data Flow Languages

Data flow visual programming languages may be classified according to their use of various design alternatives or according to their application domain. This paper groups languages by their application domain, and mentions the design alternatives which each language uses. Table 1 summarizes the design alternatives used by various languages.

2.1. Design Alternatives

The idea of using a data flow graph to represent a program has existed for some time [1]. However, the widespread use of the data flow computational model in visual programming languages is more recent [2]. The central concept of the data flow model is that a program can be represented by a directed graph where nodes represent functions and where arcs represent the flow of data between functions [3]. Arcs going into a node represent input data to a function; arcs going out represent output data—that is, the function's results. Units of data which flow on the arcs are called tokens; arcs may also be called links. Throughout this paper, the terms *arc*, *wire*, *link* and *line* are used interchangeably, as are *procedure*, *function*, *icon* and *box*.

Table 1. Design alternatives used by data flow visual programming languages (left half).

| Design Alternative | Language | | | | | | | | | |
|----------------------------|-------------|------------------------------|------------------------------|-------------------------------|------------------|-------------------------|-------------|-------------|-------------|--|
| | Hookup | Fabrik | InterCONS | HI-VISUAL | VIVA | Cantata | VIPEX | LabVIEW | | |
| Main reference | [21] | [8] | [23] | [24] | [18] | [12] | [28] | [31] | | |
| Primary application domain | Music | Constructing user interfaces | Constructing user interfaces | Image processing, office work | Image processing | Image/signal processing | Science | Science | | |
| Box-line representation | Y | Y | Y | N | Y | Y | Y | Y | Y | |
| Iteration | Y | Y | Y | Y | Y | Y | Y | Y | Y | |
| Procedural abstraction | N | Y | Y | Y | Y | Y | Y | Y | Y | |
| Selector/distributor | N | N | Y | Y | N | N | N | N | N | |
| (or derived functions) | | | | | | | | | | |
| Flow of data | Uni. | Bi. | Uni. | Uni. | Uni. | Uni. | Uni. | Uni. | Uni. | |
| Sequential execution | N | N | N | N | N | N | N | N | Y | |
| construct | | | | | | | | | | |
| Type checking | N | Y | N | N | N | Y | N | Y | Y | |
| Higher-order functions | N | N | N | N | N | N | N | N | N | |
| Execution mode | Data-driven | Data-driven | Data-driven | Data-driven | Data-driven | Both are available | Data-driven | Data-driven | Data-driven | |
| Liveness level | 2 | 3 | 2 | 3 | 4 | 2 or 3 (user-chosen) | 2 | 2 | 2 | |

Table 1. Design alternatives used by data flow visual programming languages (right half).

| Design Alternative | Language | | | | | | |
|--------------------------------|---------------|----------------------------------|----------------------------------|----------------------------------|---------------------------------|------------------------------|-------------------------------|
| | ConMan | viz | VisualToolset | PROGRAPH | Show&Tell | ESTL | DataVis |
| Main reference | | | | | | | |
| Primary application domain | [33] Graphics | [34] General-purpose programming | [35] General-purpose programming | [37] General-purpose programming | [6] General-purpose programming | [10] Adds types to Show&Tell | [11] Scientific visualization |
| Box-line representation | Y | Y | Y | Y | Y | Y | Y |
| Iteration | Y | Y | Y | Y | Y | Y | Y |
| Procedural abstraction | N | Y | Y | Y | Y | Y | Y |
| Selector/distributor | N | Y | N | N | N | N | N |
| (or derived functions) | | | | | | | |
| Flow of data | Uni. | Uni. | Uni. | Uni. | Uni. | Uni. | Uni. |
| Sequential execution construct | N | N | N | N | N | N | N |
| Type checking | Y | N | Y | N | N | Y | Y |
| Higher-order functions | N | Y | N | N | N | Y | Y |
| Execution mode | Data-driven | Data-driven | Data-driven | Data-driven | Data-driven | Data-driven | Data-driven |
| Liveness level | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

2.1.1. *Pure Data Flow Model*

The 'pure' data flow model is the data model with no added control flow constructs such as **WHILE** loops, sequential execution constructs or **CASE** statements. In the pure data flow model the sequence in which functions or nodes are to execute is not specified. Rather, when all of a node's inputs are available, the node *fires*. This corresponds to executing the function. The resulting data is then put on the output arcs, where it flows downstream to other functions. While all data flow visual programming languages are based on the pure data flow model, the following design alternatives can be used to augment the pure data flow model.

2.1.2. *Box-line Representation*

Most visual data flow programming languages use boxes to represent functions and lines to represent the flow of data between functions. An advantage of choosing this box-line representation for a visual programming language is that this representation allows the user to insert viewing monitors easily at different spots to inspect the data. However, languages are not required to use this visual representation. For example, while an early version of HI-VISUAL [4] used the usual box-line representation, a more recent version [5] uses juxtaposition of boxes, rather than a line connecting boxes, to indicate that a function is being executed.

Another variation on the idea of 'boxes are functions and lines carry data' is the use of boxes to represent such things as data items, files, iteration constructs and sequential execution constructs.

2.1.3. *Iteration*

Many data flow visual languages provide iteration. Such languages use a variety of iteration constructs. These constructs include: cycles in the data flow graph; sequential ports (objects associated with a Show and Tell iteration box [6], which recycle a single changing value through repeated executions of the iteration box; parallel ports (objects associated with a Show and Tell iteration box, which split up a collection of list into its elements, execute the iteration box once for each element and gather up the results into a collection or list); control flow constructs such as **FOR**, **WHILE** and **REPEAT** loops; and predefined functions which apply an operation to all elements of a list.

2.1.4. *Procedural Abstraction*

Procedural abstraction is available in some data flow visual languages (for example, in HI-VISUAL [4] and in Show and Tell [6]). This means that an entire graph may be considered a procedure and may be condensed into a single node. Arcs going to and from the node may then be considered arguments to the procedure. Procedural abstraction is important for two reasons. First, it saves screen space, making a visual program more compact. Second, it allows the programmer to view the program at a higher level, without numerous details.

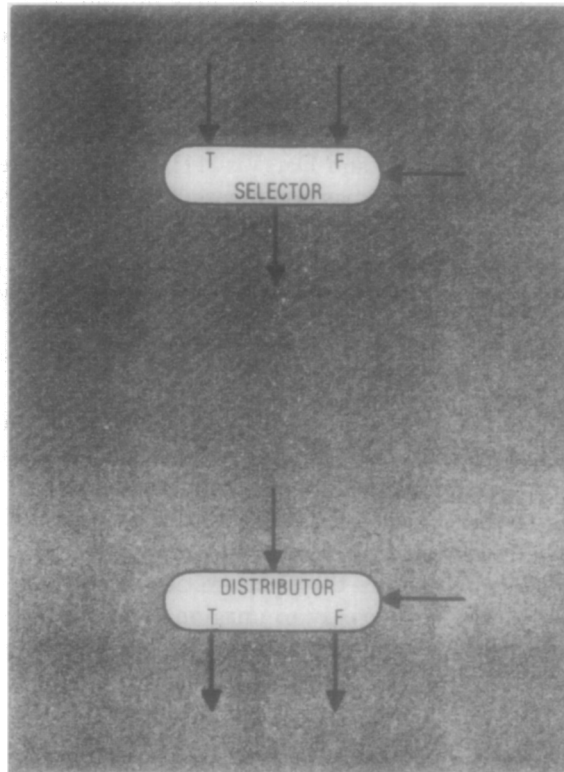


Figure 1. Selector and distributor functions [3] © 1982 IEEE

2.1.5. Selector and Distributor

Two constructs which can be added to a data flow language are the selector and the distributor (Figure 1). In its simplest form, the selector accepts a true or false data token to decide which of two inputs should be propagated to its output. The distributor uses a true or false token to pick an output arc to put its data on. Data flow visual languages rarely have either a selector or distributor in the exact form of Figure 1, but some languages have similar constructs that accomplish the same task.

2.1.6. Sequential Execution Construct

A difficulty with the pure data flow model is that it does not allow a user to specify that actions should be performed in a sequence: 'first function A, then B, then C'. Rather, in the pure data flow model, any of A, B or C could be executed as soon as its input data was available. Some languages introduce a sequential execution construct to allow functions to be executed in a specific order. Of the languages surveyed in this paper, only LabVIEW [7] possesses a sequential execution construct.

2.1.7. Type Checking

Some data flow visual languages (for example, Fabrik [8, 9]) indicate the type of data flowing over arcs, and perform type checking. Type checking is usually done at

program construction time, by not allowing the user to connect functions with types that do not match. This prevents run-time errors which would otherwise be caused by type mismatches.

2.1.8. Higher-order Functions

Some languages, such as ESTL [10] and DataVis [11], provide higher-order functions: functions which take other functions as arguments. One way to present higher-order functions is for functions to flow over arcs as data objects. If this is done, an ‘Apply’ node can then be used to apply the function to its inputs, and to produce as output the function’s result. Because it is awkward to have functions flowing on arcs, some languages instead present higher-order functions via function slots. A function slot is a box inside the icon for a higher-order function. The function slot is a parameter to the higher-order function. A lower-order function is placed in the function slot. The higher-order function (or one of the functions it calls) will eventually apply the lower-order function to some data. Higher-order functions are a useful design alternative because they add power and versatility to a language.

Figure 28 shows an example of an ESTL [10] higher-order function with a function slot. The higher-order function is ‘?’; the low-order function is square root. The higher-order function simply applies the square root to the input of nine. While this higher-order function is simple, it is easy to imagine a more complicated higher-order function: one, for example, which would apply the square root to each element of a binary tree.

2.1.9. Execution Modes

There are two possible modes of execution for a data flow program: data-driven and demand-driven. For data-driven execution, a function node executes when it receives data from upstream nodes on its input arcs, and it then sends data to nodes downstream on its output arcs. (Upstream nodes are found by following input arcs from the node backwards; downstream nodes are found by following output arcs forward out of the node.) With data-driven execution, a node fires as soon as its inputs are available, and all data flows downstream. A weakness of data-driven execution is that some nodes may fire needlessly, even though the final output of the program does not make use of their computations [12]. A language designer may decide to cure this difficulty by using demand-driven execution.

For demand-driven execution, execution of a node happens when downstream nodes request data from a node’s output arcs. The node may request data from upstream nodes over its input arcs; if it does so, it then waits for these nodes to send it data. It then sends data over its output arcs to downstream nodes.

With demand-driven execution, a node fires only when downstream nodes request data from it. All data still flows downstream, but requests flow upstream. In Cantata [12, 13], a user may choose demand-driven execution and may then select an icon which displays the result of a program on the screen. Cantata then looks upstream from the display icon and finds and fires only the nodes that must be executed to produce the output. This ‘winnowing’ of icons to be run may save significant time at execution. Demand-driven execution, due to its greater complexity, is used less often than data-driven execution. In fact, all the visual languages which this paper surveys, except Cantata, use data-driven execution.

2.1.10. *Level of Liveness*

Tanimoto has described a four-level scale of liveness [18]. The first, 'informative' level uses the visual representation of the program for documentation or education about the program, but not for instructions to the computer. A flowchart for a FORTRAN program is at this level.

The second, 'informative and significant' level uses the visual representation as the program. The representation resides on the computer and is executable. Once the user has entered the program into the computer, he or she may tell the computer to execute it. PICT [19] is a system at this level.

The third, 'informative, significant and responsive' level includes systems that execute automatically whenever the user edits the program or its input data. The user does not have to give an 'Execute' command. HI-VISUAL [20] is at the third level.

The fourth level is 'informative, significant, responsive and live'. A system at this level continually updates the display to show the results of processing data. VIVA [18] is a system at the fourth level.

Unless otherwise specified, the languages in this paper are at the second level of liveness.

2.2. **Application Domains**

The next section examines 15 visual programming languages that use the data flow computational model. The languages are: Hookup, Fabrik, InterCONS, HI-VISUAL, VIVA, Cantata, VIPEX, LabVIEW, ConMan, *viz*, Visual ToolSet, PROGRAPH, Show and Tell, ESTL, and DataVis.

The languages are organized by their application domains, which include music, operating system commands, construction of user interfaces, image processing, science, graphics, general-purpose programming and languages related to Show and Tell.

Hookup deals with music; Fabrik and InterCONS are for constructing user interfaces; HI-VISUAL, VIVA and Cantata are intended for image processing; (more recent work on HI-VISUAL is oriented toward office work; Cantata has also been used for signal processing and other applications); VIPEX and LabVIEW deal with science; ConMan is for graphics; *viz*, the Visual ToolSet, PROGRAPH and Show and Tell are general-purpose programming languages; ESTL adds types and higher-order functions to Show and Tell; and DataVis deals with scientific visualization and is in the same family of languages as Show and Tell.

AVS [15, 16] and apE [17], while useful data flow systems for visualization of scientific data, are not included in this paper because they are not visual programming languages. That is, one cannot write a new library function while staying solely in the visual portions of AVS or apE. Instead, one must switch to a different, textual, language (C or FORTRAN) to define the new function.

3. **Survey of Various Languages**

3.1. **Languages For Music**

3.1.1. *Hookup*

Hookup [21] is a low level data flow visual language for controlling music synthesizers. Hookup provides various predefined library functions which may be

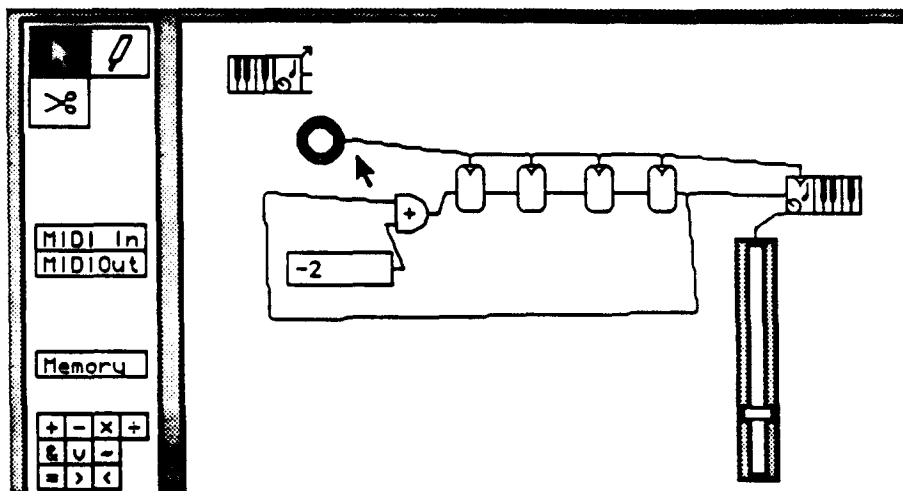


Figure 2. Simple Hookup program [21]

called by a programmer. Library functions are represented by icons. Although Hookup uses the typical box-line representation in which icons represent functions, icons can also represent input/output devices. Arcs transmit simple data values such as integers or the value of a musical note (an integer). Hookup's library functions do simple arithmetic and provide user interface features such as buttons and sliders. The 'MIDI In' and 'MIDI Out' icons handle input and output for the synthesizer. Memory cells store a single integer.

Two other interesting capabilities of Hookup are its Vector data type and the Graph icon. Icons (functions) are available to give the length of a vector, the n th element of a vector, and whether or not a value is an element of the vector. The Graph icon continually plots points at given X and Y coordinates, and has a trigger input (that is, a triangle) to erase itself.

Figure 2 shows a simple Hookup program. The four memory cells hold four notes, played in earlier from the MIDI keyboard via the now-disconnected MIDI In icon at the top left. Each time the round button is pressed, the series of four notes is sent to the synthesizer to be played, via the MIDI Out icon. The slider controls the volume. Since the memory cells send their contents to the right when triggered by a button press, adding -2 ensures that the melody will descend by two half steps each time it is played.

Hookup provides iteration in the form of cycles in the data flow graph. The program in Figure 2 can be changed by replacing the button with a clock which is frequently reset to 0. Once this is done, continuous iteration will occur, rather than one execution of the circuit each time the user presses the button.

3.2. Languages for Constructing User Interfaces

3.2.1. Fabrik

Fabrik is a language for constructing user interfaces [8, 9]. In Fabrik, icons are square boxes and represent functions. Functions are called components. Fabrik provides a set

of predefined components and programmers may also construct their own components. Their predefined components perform arithmetic, string handling and graphic manipulation. In addition, the components generate graphemes, which include rectangles, ovals, lines, polygons or bitmaps. Another group of predefined components provides transformations of graphemes: scaling, rotation, translation, hiding and inverting a grapheme and merging several graphemes. Another group of components represents user interface items (for example, the mouse).

Arcs or wires are used to connect components; data flows over the wires. Wires are not connected directly to components; rather, they are connected to pins on a component. Pins serve three purposes: they indicate how many wires may connect to a component; they indicate where on a component's periphery a connection may be made; and they indicate whether the connection is for input, output or both. As can be seen in Figure 3, output pins point away from a box (the 9/5 box in Figure 3), input pins point toward a box (that is, from the 9/5 box toward the \times/\div box in Figure 3) and input/output pins are diamonds (the left and right sides of the $+/-$ box in Figure 3).

Input/output pins are necessary because Fabrik allows bi-directional data flow. On links connecting diamond-shaped pins, data may flow either way. The direction of flow depends on where the input is coming from at any given time. In effect, Fabrik is using one bi-directional link to represent two separate uni-directional links. Bi-directionality allows a function and its inverse to be included in the same component. For example, the $+/-$ box in Figure 3 can either add or subtract, depending on which side it receives a new input value. Note that 9/5 and 32, however, are constants. They would never be changed to satisfy constraints in the program because their pins are not bi-directional.

Fabrik provides type checking. Each predefined component assigns a type to each pin. Thus, input and output are restricted to certain types. When the programmer

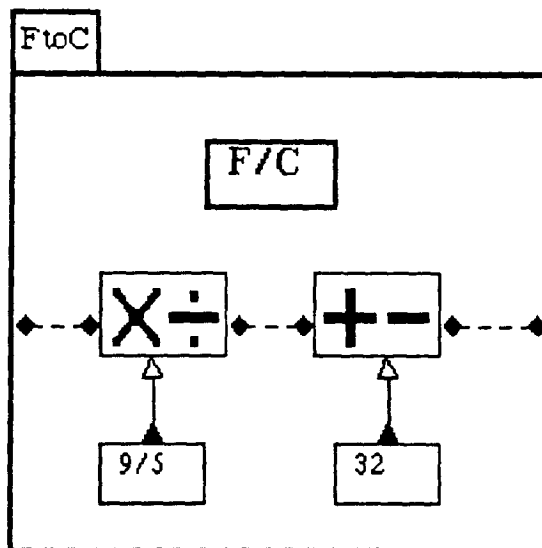


Figure 3. Fabrik internal diagram for a Fahrenheit/Centigrade component [8]

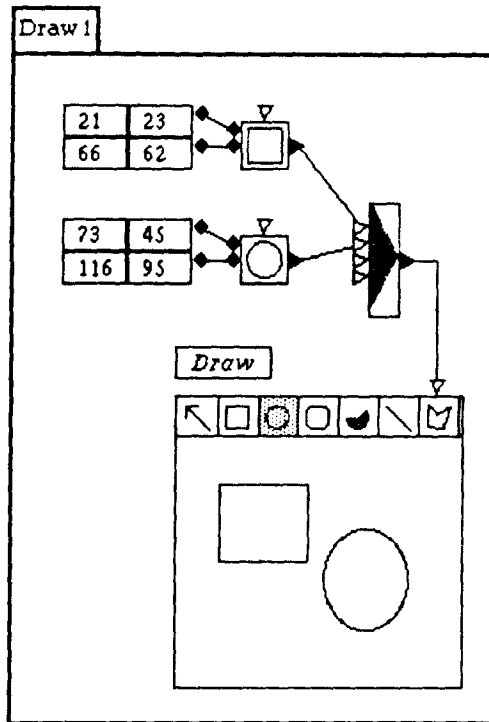


Figure 4. The Fabrik Draw component automatically lays out diagrams as the user creates a drawing [8]

attempts to wire together two pins, Fabrik does type checking to verify that the two types match. If they do match, the connection is made. If not, the connection is not made and an error message is shown.

Fabrik's types are: primitive (this includes number, Boolean, character and grapheme), bundled (like a record type, this type may contain different types within itself), array and enumerated.

Several other features of Fabrik are of interest. A **Draw** component (Figure 4) functions somewhat like MacDraw [22]. The user creates a structured drawing, and **Draw** creates the Fabrik program which would generate the drawing.

Fabrik provides iterators for computations which are to be repeated. Iterators and other components transfer data across their own boundaries to or from the outside by using special pins called gateways. Some gateways allow an iterator to receive a collection of elements. An inbound collection gateway splits up the collection and fires the iterator once for each element in the collection. A corresponding outbound gateway gathers up into a new collection the data values produced by each firing of the gateway. Fabrik's [8] approach to iteration was inspired by Show and Tell [6]; gateways which deal with collections of elements are similar to parallel ports in Show and Tell [6].

3.2.2. InterCONS

InterCONS is a language for creating user interfaces [23]. InterCONS has boxes for various purposes and links to carry data between boxes. Boxes may be constants,

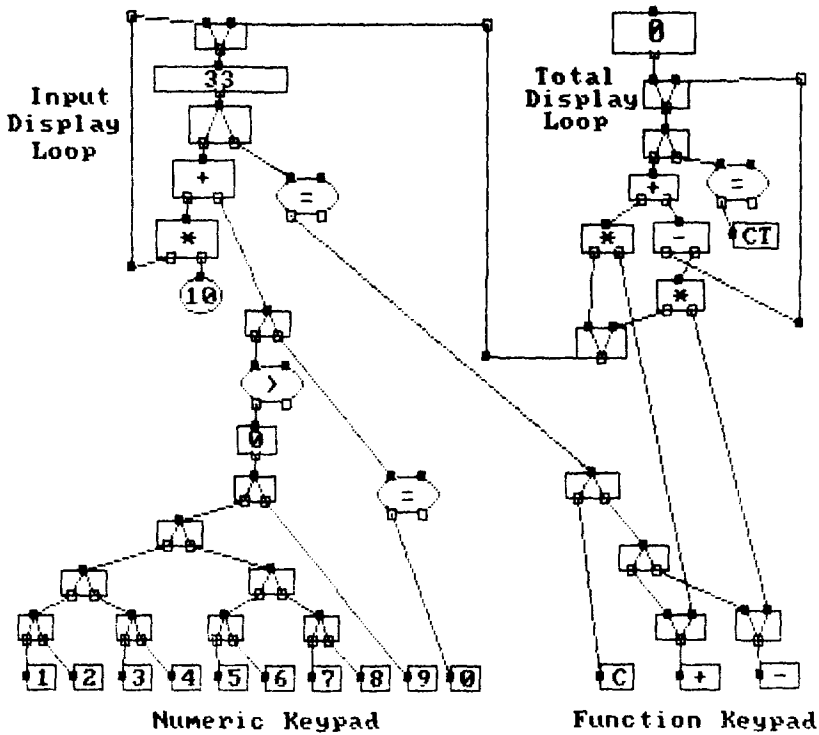


Figure 5. InterCONS program for calculator internals [23] © 1988 IEEE

variables, text or functions. Among the predefined library functions provided are arithmetic operators, min, max and relational operators. Predefined boxes for user interface elements like sliders, buttons and gauges are present. Fan In and Fan Out boxes merge and split links. A 'path splitter' box functions like a distributor function. If its condition is true, its input is shunted out one of its two outputs; if the condition is false, the input goes to the other output. Figure 5 shows an InterCONS program to generate a calculator.

InterCONS provides iteration in the form of cycles in the data flow graph. Every cycle has at least one button connected to it to control its execution. When the user presses and releases the button, the cycle executes once; if the user presses and holds the button, the cycle executes repeatedly.

InterCONS allows procedural abstraction by permitting the programmer to specify that certain boxes in an InterCONS program can be joined together to form a single small box (an icon). This icon may then be used as a component in a larger program.

3.3. Languages for Image Processing

3.3.1. HI-VISUAL

HI-VISUAL is a visual data flow language that emphasizes icons [4, 5, 20, 24–26]. Its application area was originally image processing, but now also includes work done in an office environment: simple accounting, word processing, report generation, data storage and retrieval.

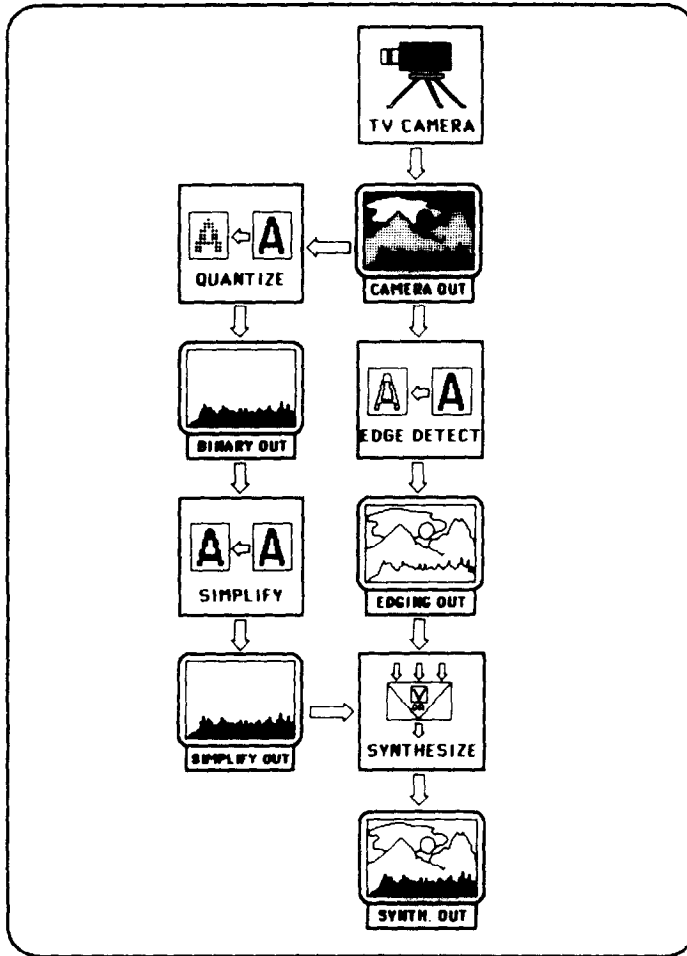


Figure 6. A complete HI-VISUAL program [4]

In HI-VISUAL, icons represent functions. The flow of data is indicated by arrows. As well as representing functions, icons also represent the input data to a procedure and the output data from a procedure. Figure 6 shows a HI-VISUAL program. (This is an early version of the language.) HI-VISUAL provides procedural abstraction, since one icon can represent an entire function. Figure 7 shows the function behind the QUANTIZE icon in Figure 6.

HI-VISUAL provides two iteration constructs. Both are control flow constructs. The first is akin to a FOR loop; it allows the programmer to specify that, for example, certain icons should be executed n times, but it does not allow access to the loop control variable. The second iteration construct is a REPEAT-UNTIL loop.

A switch statement is available. The switch functions like a CASE statement, but controls the flow of data rather than the flow of control. Depending on the value of a Boolean expression, data coming into the switch is routed to one of two sets of outgoing links. There may also be multiple incoming links to the switch. This switch is akin to the distributor function discussed in Section 2.1.

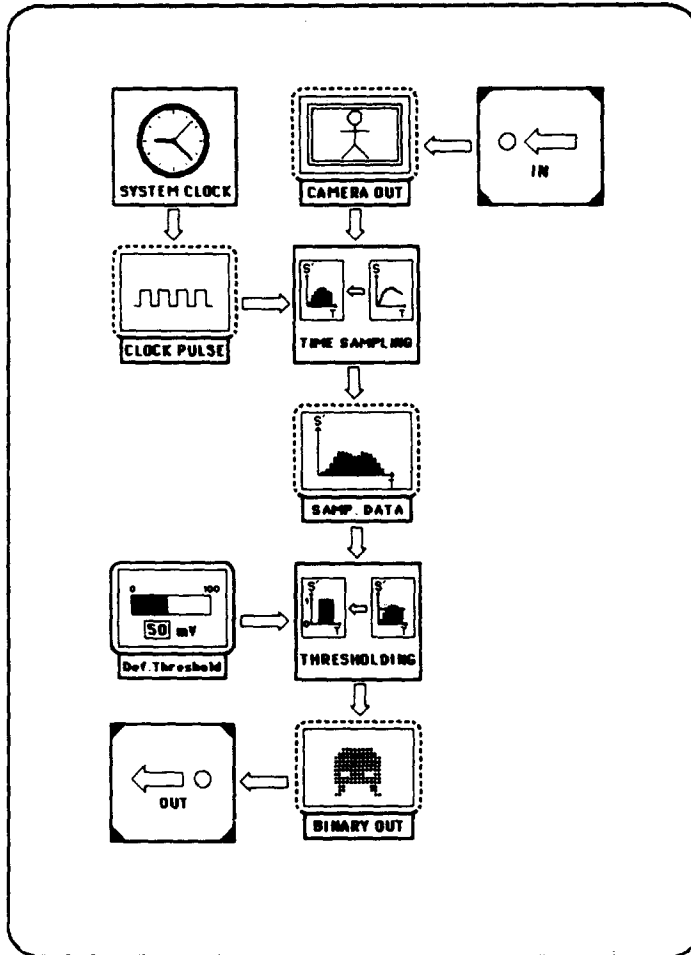


Figure 7. A lower-level HI-VISUAL function behind QUANTIZE in Figure 6 [4]

In a more recent version of HI-VISUAL [5], icons are used only for data objects, not for functions. Also, the approach to the system has become object-oriented. Since data objects are considered to 'own' operations (in accordance with the object-oriented paradigm), icons for functions are no longer shown.

Links are now used only for indicating that output data is coming from a function execution. Such a function execution happens when two data icons are superimposed. A data icon's behavior is not fixed, but depends on what icon it overlaps. If a second icon overlaps the first data icon, this also affects the first icon's behavior. Each icon is now considered an object, an instance of a class. Through class membership, an icon inherits class variables, instance variables, a set of acceptable messages and a set of transmittable messages. Acceptable messages are messages that an icon can receive and execute (that is, functions the icon can do) and transmittable messages are messages the icon can send.

A database of rules is kept. When the user overlaps two icons the rules determine which icon is the passive object (to be checked for acceptable messages) and which

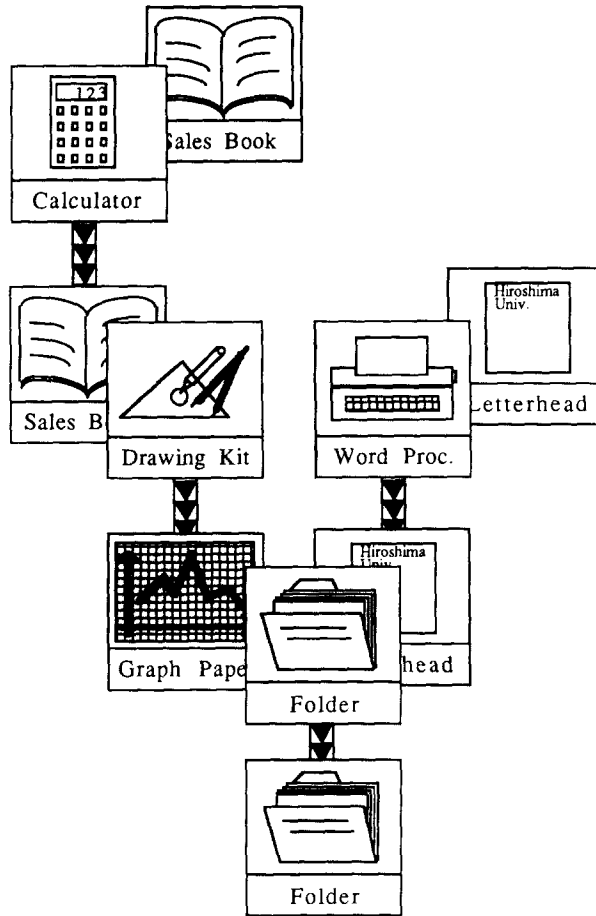


Figure 8. A more recent HI-VISUAL program [5] © 1988 IEEE

icon is the active object (to be checked for transmittable messages). If there is not an immediate match between a transmittable message and an acceptable message, a search up the inheritance hierarchy (that is, into superclasses) is done.

Figure 8 shows a more recent HI-VISUAL program. When the user superimposes the calculator over the sales book at the top left, a new sales book is produced. The drawing kit can then be applied to it to produce a graph paper, and so on.

In a more recent version of HI-VISUAL [25], the user interface has hidden the idea that HI-VISUAL is based on data flow. (See Figure 9.)

3.3.2. VIVA

VIVA is a language for image processing [18, 27]. Boxes represent functions; links transmit data between boxes. Three kinds of boxes are available to the programmer: sources, operators and monitors. Sources provide image data, while monitors display images and output from operators. Operators include mathematical and image-processing functions. Operators may be subprograms. Control panels may be defined

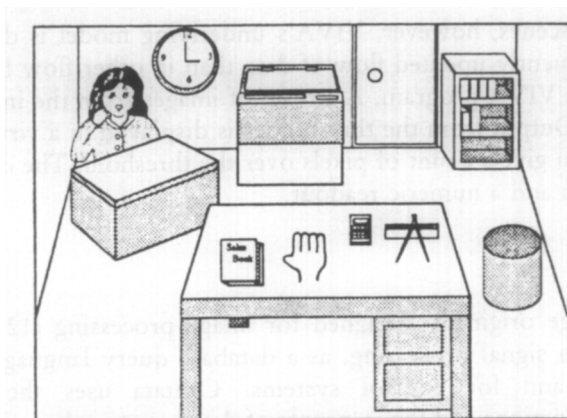


Figure 9. User interface of HI-VISUAL'89 [25] © 1989 IEEE

for sources, operators and monitors, and provide graphical input and output indicators like switches and meters. Liveness is a goal of VIVA. Not only does VIVA try to execute the program whenever the user edits it, but VIVA frequently updates its display to show the new streams of data being calculated. VIVA is at the fourth level of liveness on Tanimoto's scale. Iteration is present in the form of a cycle in the data flow graph, rather than a control flow iteration construct.

VIVA does not have an explicit data flow model. Since VIVA is designed to achieve a high degree of liveness, VIVA's metaphor is a continuously updated electronic

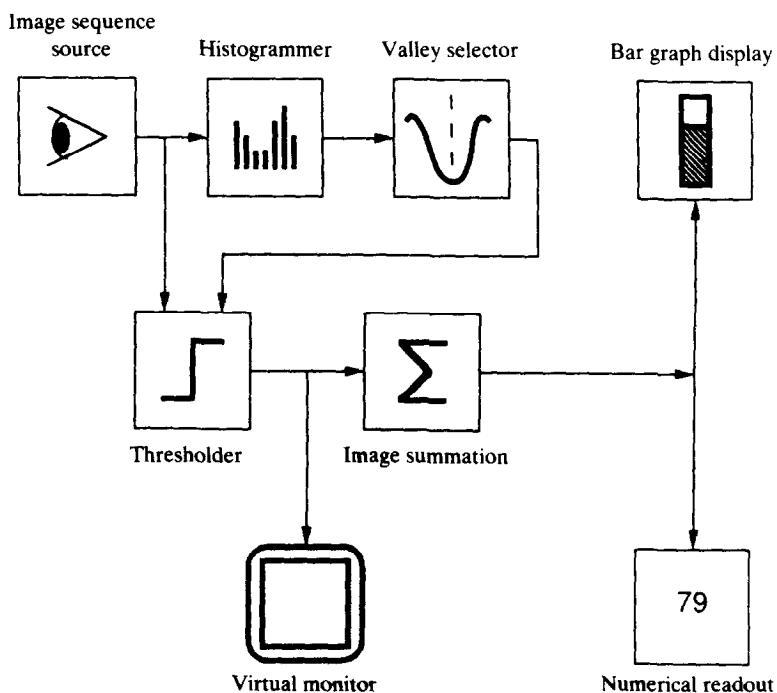


Figure 10. A VIVA program for image stream thresholding and display [18]

circuit. Behind the scenes, however, VIVA's underlying model is data flow, albeit a more rapid and frequently updated flow of data than in other flow languages.

Figure 10 shows a VIVA program. A stream of images from the image source is fed to the thresholder. Output from the thresholder is displayed in a virtual monitor, and also is summed up to give a count of pixels over the threshold. The count is displayed by using a bar graph and a numeric readout.

3.3.3. Cantata

Cantata is a language originally designed for image processing [12–14]. It has also been found useful in signal processing, as a database query language, for expressing matrix algorithms and for control systems. Cantata uses the usual box-line representation for functions and the transport of data between them. The boxes, called glyphs, may be selected from a library of approximately 240 algorithms for signal and image processing.

Two forms of iteration are available. The two forms are control flow constructs: a **COUNT-LOOP** (similar to a **FOR** loop) and a **WHILE-LOOP**. Each of these loops is represented by an icon which is placed on a cycle in the data flow graph. When a condition in the loop icon is satisfied, data can flow through the exit path from the cycle.

Figure 11 is a Cantata program containing a **COUNT-LOOP**. The program averages a sequence of five noisy images. The five small images at the bottom are the result after each iteration. The large image at the top is the final result.

An **IF-THEN-ELSE** construct controls the flow of data, rather than the flow of control. A conditional expression is evaluated to give **TRUE** or **FALSE**. This result is then used to direct data down one of two possible paths. This construct is a form of

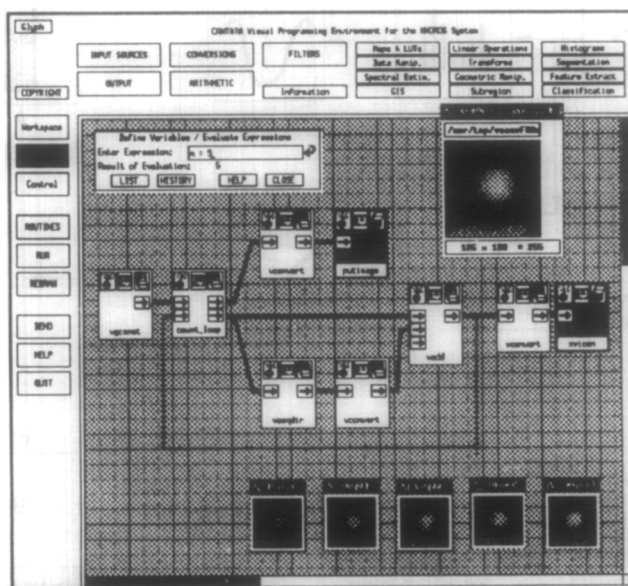


Figure 11. Cantata program showing **COUNT-LOOP** iteration construct [13] © 1990 IEEE

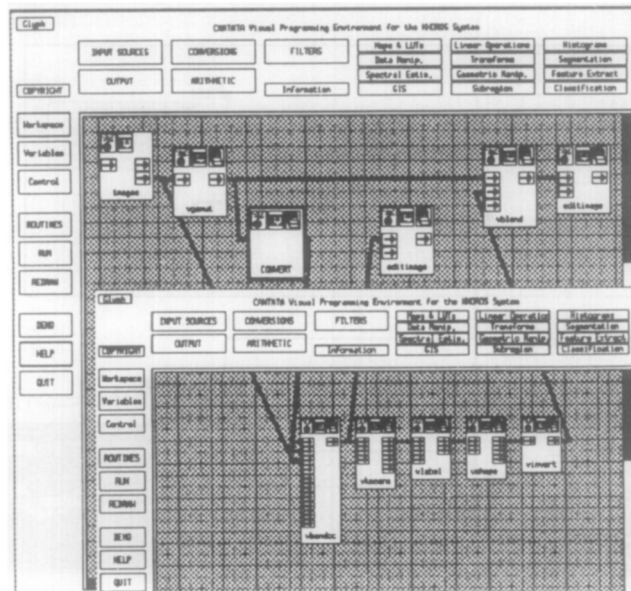


Figure 12. Close-up view of a Cantata function [13] © 1990 IEEE

the distributor function (discussed in Section 2.1), and is also similar to HI-VISUAL's switch statement.

Cantata provides procedural abstraction. It is possible to zoom in on a function icon in order to see its internal structure. This has been done in Figure 12. One can think of zooming in on a function as 'moving down' one level in the visual program. Functions at the lowest level (that is, those not composed of other function icons) are written in textual C or FORTRAN.

Cantata uses a hybrid of data-driven and demand-driven execution. Typical execution in Cantata is data-driven: a function may execute as soon as its input data is available, and data flows downstream. However, a Cantata user may choose the option of demand-driven execution. The user can select an icon which displays data generated by a program; only upstream icons which are necessary to produce the displayed data are then executed.

Cantata's liveness level is normally two—that is, a RUN command must be given for a program to execute. However, a user may choose responsive mode, which makes Cantata's liveness level become three; whenever the user edits the program or provides new input data, the program automatically executes.

3.4. Languages for Science

3.4.1. VIPEX

VIPEX [28, 29] is a language for performing scientific visualization [30]. The current application domain is image processing, which is a part of scientific visualization. VIPEX, like most data flow visual languages, has boxes and lines. There are three kinds of boxes: processing objects, composition objects and group objects. They can be distinguished by the shape of the icon at their top left corner. In Figure 13, a

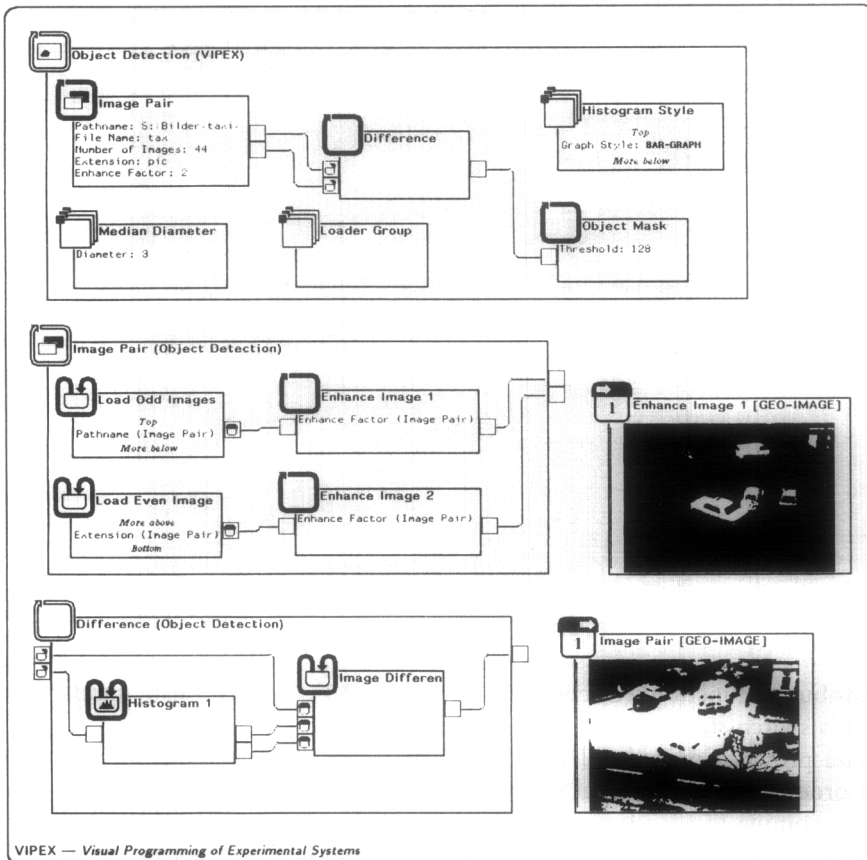


Figure 13. VIPEX functions and data objects [28] © 1988 IEEE

sample VIPEX program, Load Odd Images (inside Image Pair) is a processing object, Object Detection is a composition object, and Median Diameter (inside Object Detection) is a group object.

A processing object is a low level function; it calls no other (VIPEX) functions. Processing objects consist of textual Lisp code. A composition object is a VIPEX function that calls other VIPEX functions. Composition objects provide procedural abstraction in VIPEX. A group object is used for such purposes as propagating a global parameter value to several functions which do not necessarily call each other. A group object does not provide procedural abstraction. It does not necessarily call the functions inside it.

With regard to program execution, each box can have one of three states: (1) stopped (the function is not firing at all); (2) step (when the function's input plugs have data and its output plugs are empty, the function fires once and stops); or (3) cycle (under the same conditions as for 'step', the function fires repeatedly). The 'cycle' state is the way in which VIPEX provides iteration. At present, VIPEX does not possess higher-order functions. The creators of VIPEX, however, have mentioned such functions as a possible extension to VIPEX [29].

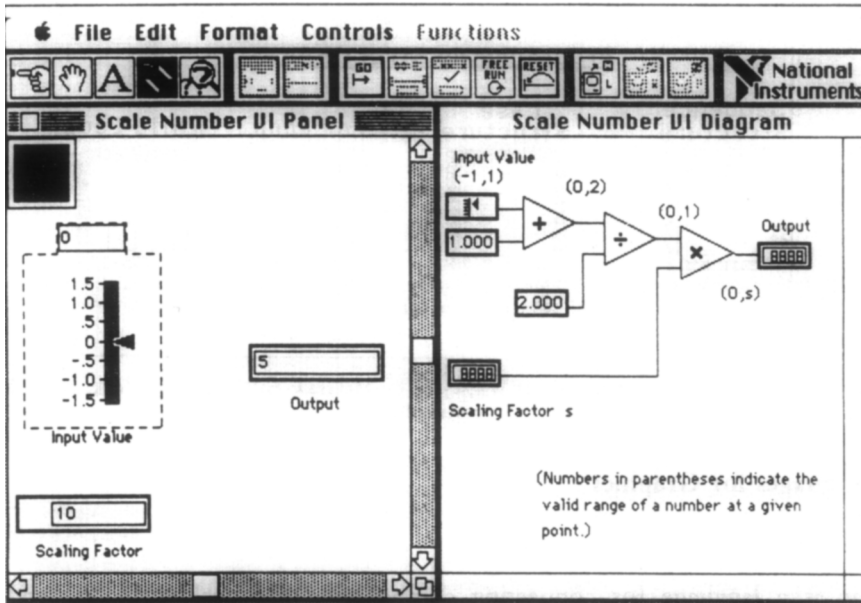


Figure 14. The LabVIEW Scale Number virtual instrument. The left half of this figure contains the front panel for this instrument: it contains two input variables, or controls, and one output control. The right half is the block diagram [32] © by McGraw-Hall, Inc. New York, NY. All rights reserved

3.4.2. LabVIEW

LabVIEW is a high level language for use in data collection from lab instruments and for use in data analysis [7, 31, 32]. It is a commercial product. Program modules (procedures) are called virtual instruments. Each instrument has two parts: a front panel on the left and a block diagram on the right (Figure 14). The front panel is the user interface to the procedure. It contains input and output data (numbers) for the procedure, as well as sliders, knobs, strip charts, meters and x - y graphs. Its purpose is to display graphically input to and output from the procedure. The block diagram is the data flow visual program for the procedure.

LabVIEW provides a broad selection of predefined function boxes. These functions include arithmetic operations, trigonometric functions, string manipulation routines, statistical analyses, signal processing, matrix operations and curve fitting. Programmers can also define their own functions.

Various boxes representing control flow structures are introduced into the pure data flow computational model (Figure 15). These boxes include a **FOR** loop, a **WHILE** loop, a **CASE** statement and a sequence structure. Despite the **WHILE** loop's name, it is really a **REPEAT...UNTIL** loop. Its contents are always executed at least once. The sequence structure forces sequential execution of a series of data flow diagrams; such explicit indication of 'do diagram A, then B, then C' is normally lacking in data flow programs.

LabVIEW's data types include Boolean values and arrays of Booleans, real numbers and arrays of reals, strings and arrays of strings, and structures (similar to structures in C).

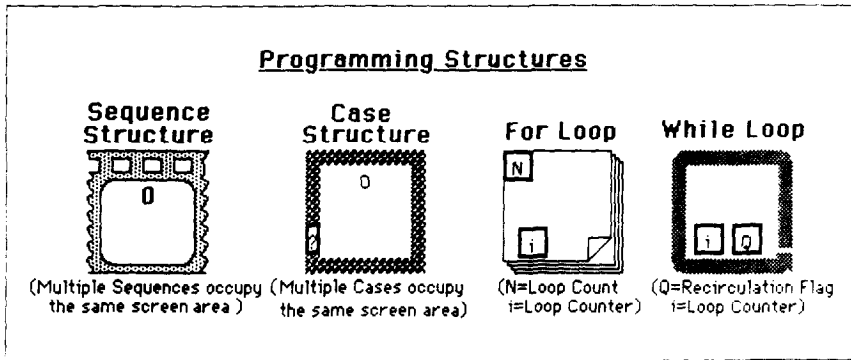


Figure 15. Control flow structures in LabVIEW [32]. © by McGraw-Hill, Inc. New York, NY. All rights reserved

3.5. Languages for Graphics

3.5.1. ConMan

ConMan is a language for connecting components that deal with 2-D and 3-D graphics [33]. ConMan is not a complete general-purpose programming language. Rather, it allows a programmer to connect prebuilt components together. These components include 2-D and 3-D interactive line editors, a color 3-D hidden-surface-removal renderer, a paint program and a tape recorder. The user connects components by joining them with lines. Over the lines flow data values, which are restricted to be integers and real numbers.

Figure 16 shows a simple ConMan program. A 2-D curve editor is connected to a 3-D sweep editor. Two view editors (specifying rotation, transformation, and/or

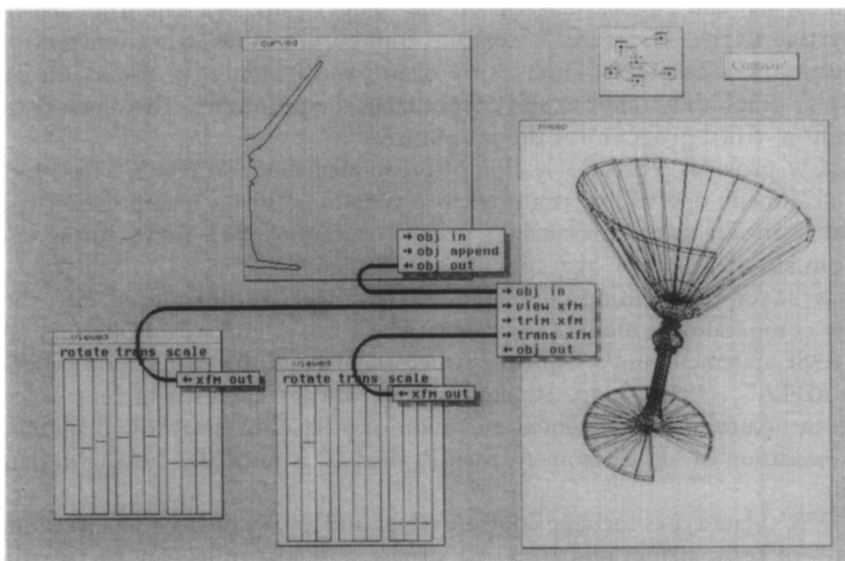


Figure 16. A simple ConMan program [33]

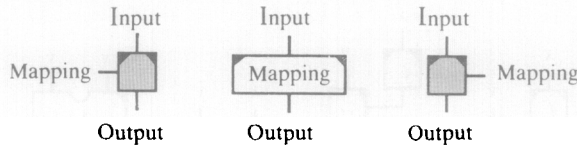


Figure 17. Visual representation of functions in viz [34] © 1990 IEEE

scaling) are connected to the sweep editor. One controls the view of the swept surface; the other provides a transformation that is repeated to create the swept surface. If the tape recorder were inserted into the program, it could record a series of viewing transformations or geometric objects. Such a series could then be played back.

ConMan provides a kind of implicit iteration which can be used to rotate repeatedly a 2-D curve around an axis in three-space to sweep out a 3-D surface. Rather than specifying the iteration via a data flow cycle or a control flow construct, the user simply indicates that the 2-D curve is to repeatedly undergo a geometric transformation (scaling, rotation or translation) in three-space. After each such transformation, the curve is redrawn at its new location. Eventually a complete 3-D surface is created. This kind of iteration is used in Figure 16. Available types of data include bitmap images, RGB colors, transformations and geometric shapes.

3.6. General-purpose Programming Languages

3.6.1. Viz

Viz is a general-purpose programming language [34]. It is intended to allow a representation of a functional semantics, and to be expandable to more general semantic models.

Boxes represent function applications. As Figure 17 shows, input to a function goes into the top of a box, output comes from the bottom and the interior of either side of the box is associated with the function (called a mapping). ‘Dog-ears’ on the top of a box indicate a normal function, while dog-ears on the bottom indicate the inverse of a function.

Composition of functions is done by using a line to connect an output to an input or a mapping. The boxes of the two functions involved may either be juxtaposed or may be connected by a line. The default direction of data flow is down. Thus Figure 18 shows $\lambda y . \lambda z . x(yz)$.

A semi-circular indentation in a function’s box indicates that the function has a sequence of arguments or results. The comma operator constructs sequences. Since sequences are not commutative, a single dog-ear indicates the first argument. ‘Later’ arguments are progressively farther from the dog-ear. All of the programs in Figure 19 have the same meaning.

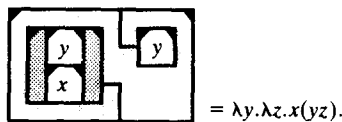


Figure 18. $\lambda y . \lambda z . x(yz)$ in viz [34] © 1990 IEEE

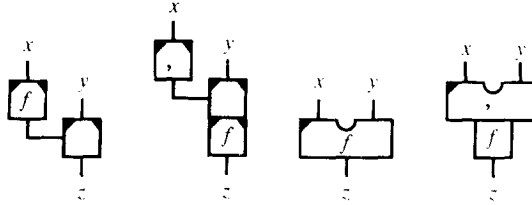


Figure 19. Different ways of expressing a function application in *viz* [34] © 1990 IEEE

Viz provides three conditional constructs. The *else* function, shown in Figure 20, returns its first argument if it is non-empty. Otherwise, *else* returns its second argument. *Else* is similar to the selector function (described in Section 2.1.).

A horizontal line with two upper ends tests for equality. Such a line is part of the *else* construct in Figure 20. If the two upper values are the same, the line takes that value. Otherwise, the line's value is \perp , the empty value.

The *if* function returns its first argument if its second argument is non-empty. Otherwise, the *if* function returns \perp .

Viz permits cycles in the data flow graph. These cycles may be used for iteration, although *viz* does not provide a construct to terminate a data flow cycle which is used for iteration.

In addition to providing data flow cycles, *viz* has another form of iteration: a functor, which applies a function to all elements of a structure (for example, a sequence) and then combines the results to make a new structure. This iteration construct is similar to Show and Tell's parallel ports. Such a function can also be viewed as a limited form of a higher-order function.

3.6.2. Visual ToolSet

The Visual ToolSet [35, 36] is a general-purpose programming language. It includes tools for defining data objects and data types, defining functions, managing a database, using control flow and programming by example. The Function Definition Tool (FDT) is of interest in this paper, because this tool is a data flow visual programming language.

Boxes in FDT represent data objects, functions and control structures. Links may be used to carry data between boxes. Alternatively, function boxes may be juxtaposed vertically or horizontally. Vertical adjacency represents function composition (that is, the output of the upper function is the input of the lower function), while horizontal adjacency represents construction (that is, both functions use the same inputs).

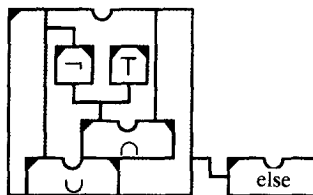


Figure 20. The *viz* *else* function [34] © 1990 IEEE

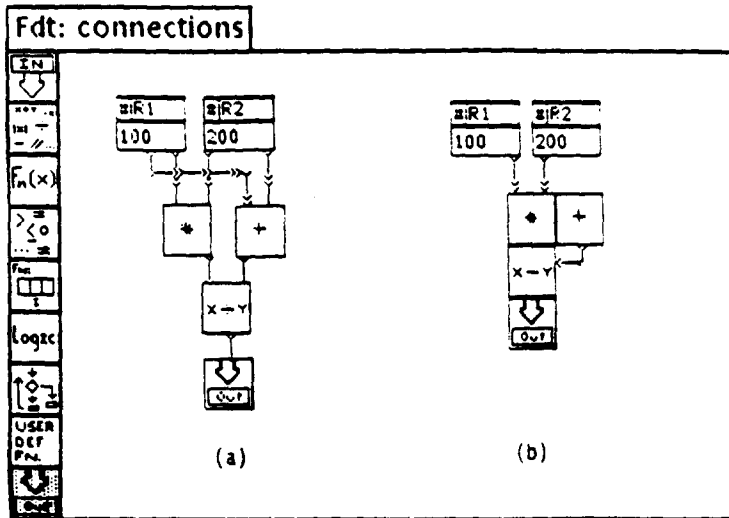


Figure 21. Data flow connections in the Visual Toolset [35]

Figure 21 shows two versions of an FDT program to produce the ratio of the product to the sum of 100 and 200. Figure 21(a) illustrates the use of links, whereas Figure 21(b) emphasizes juxtaposition.

In FDT, data is typed. Available types include primitive types and more complex types such as arrays and records. FDT provides a variety of predefined arithmetic, relational, logic and mathematical functions. Some of these functions may be modified by specifying that their inverse or reduction is desired. The reduction function modifier uses the function to reduce an array to a single value. For example, using the reduction function modifier with + on an array would mean that + should be applied to all elements of the array to yield a single sum.

Iteration boxes provide several iteration constructs. An iteration box evaluates a function for each member of an array. Options available include collect, select and reject. All of the options apply the function to each element of the array. Collect returns an array with the results of each application; select returns an array of the elements for which the function evaluates to true; reject returns the elements for which the function returns false. Collect, select and reject are akin to Show and Tell's parallel ports (described in Section 3.6.4.), which iterate over all the elements of a collection or array. In addition to the previous iteration constructs, FDT also provides a control flow construct for iteration: a **WHILE** loop.

Figure 22 shows an FDT function with an iteration box that uses select. The function accepts an array and returns an array of the input values that are integers. Each element whose ceiling equals its floor is selected. FDT also provides constructs for **IF ... THEN ... ELSE** and **CASE** statements.

3.6.3. PROGRAPH

PROGRAPH is a general-purpose programming language [37, 38]. PROGRAPH combines the data flow model with object-oriented programming. Functions are

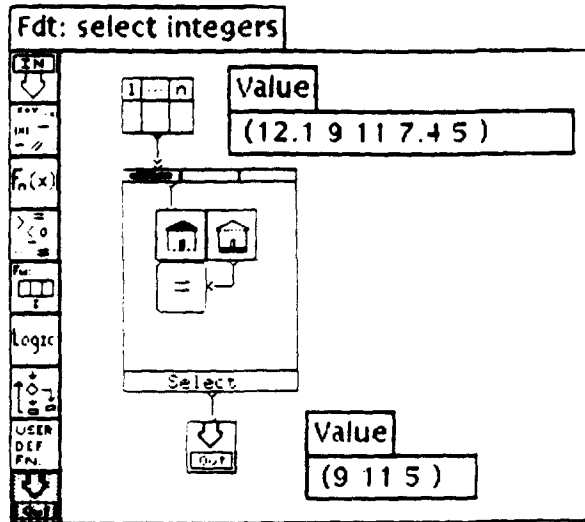


Figure 22. Example of select iteration in the Visual Toolset [35]

called methods and belong to classes. Instance variables, class variables and inheritance of variables and methods are all present.

A method is a data flow diagram. The diagram may be condensed into a single icon. Figure 23 shows class Index and two of its methods. Iteration is available. Two kinds of iteration constructs are provided: functions that apply an operation to all elements of a list, and return a list; and a control flow construct—a **WHILE** loop. The functions which are the first kind of iteration construct are similar to FDT's collect (in the Visual Toolset).

3.6.4. Show and Tell

Show and Tell is a general-purpose programming language [6, 39–41]. Its intended users are school children.

As in most visual data flow languages, boxes represent functions/procedures. However, in Show and Tell, boxes may also be constants, variables, containers of inconsistency, iterators, files and records. Links between boxes carry integers, real numbers and files. A Show and Tell variable is simply an empty box capable of holding a number. A constant is a variable that has a value at the start of execution, and that may not be changed.

Inconsistency is a key concept in Show and Tell. Inconsistency is a combination of

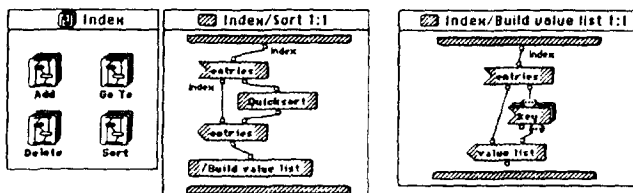


Figure 23. PROGRAPH: class Index and two of its methods [38] © 1989 IEEE

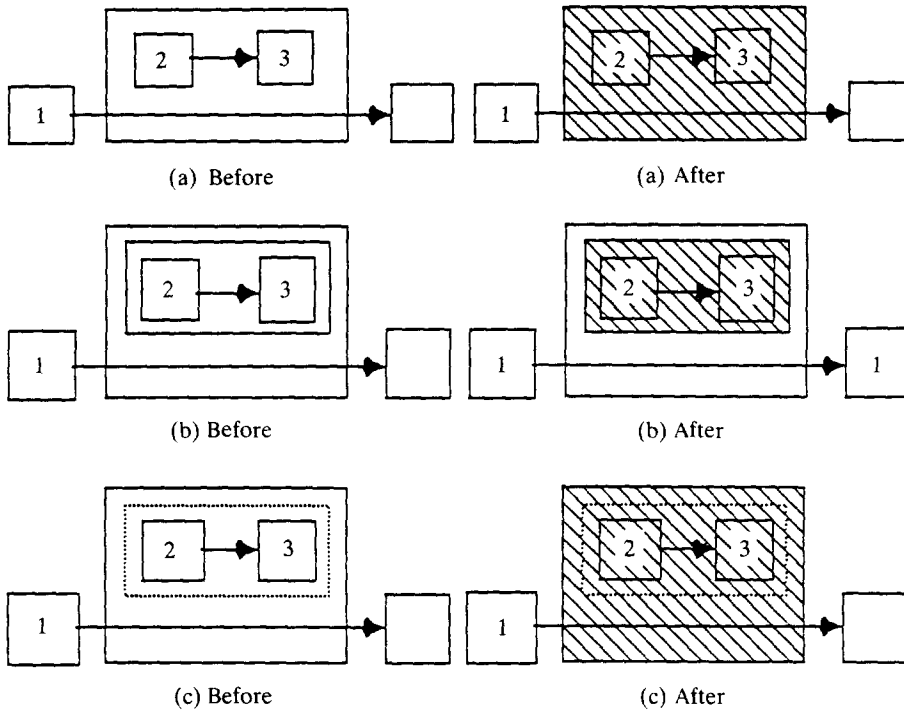


Figure 24. Show and Tell: Switching with consistency [6]

IF-THEN statements and a Boolean type. Data flow may not continue through an inconsistent box. Cross-hatching (parallel diagonal lines on top of a box) indicates inconsistency. Boxes may become inconsistent in two ways: two different values flow into a variable box, or an open box containing a predicate (like '<') is evaluated to be false. Closed (solid) boxes contain inconsistency, while open (dotted) boxes allow inconsistency to propagate out to the next larger containing box. Figure 24 shows how consistency affects data flow on links.

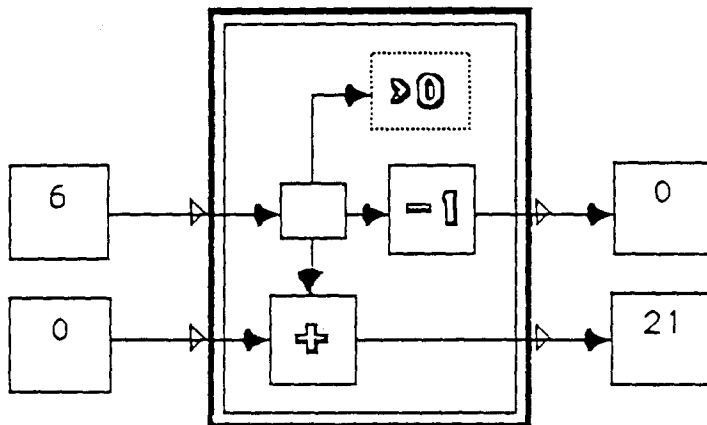


Figure 25. Show and Tell: bounded sequential iteration [6]

Show and Tell provides iteration through iteration boxes. Figure 25 shows an iteration box. An iteration box stops iterating when it becomes inconsistent. In Figure 25 this will happen when the variable in the iteration is no longer greater than zero.

Two kinds of ports can be attached to iteration boxes: sequential ports and parallel ports. These ports in Show and Tell are different from Fabrik's pins. Show and Tell's ports are not for input/output with all boxes, but only for iteration and file boxes.

Sequential ports are used for recycling a single data value from one iteration to the next. The value may be changed during any of the iterations. A sequential port is analogous to the variable X in the Pascal code

```
FOR I := 1 TO 10 DO
  X := X + I;
```

Sequential ports are always used in pairs with one port on each side of the iteration box. Thus, in Figure 25 two pairs of sequential ports are used. The top pair helps give a value to the counter variable, and the bottom pair helps with the sum variable.

Parallel ports are used for transferring a whole series or collection of data values between boxes. A parallel port is analogous to the array **INFO** in the Pascal code

```
FOR I := 1 TO 10 DO
  X := X + INFO[I];
```

where **INFO** is a 1-D array. Figure 26 shows two iteration boxes connected by parallel ports. When the top box becomes inconsistent, iteration in both boxes ceases.

3.7. Languages Related to Show and Tell

3.7.1. Extended Show and Tell

Najork and Golin [10] have described Extended Show and Tell (ESTL), which adds a type system and higher-order functions to the Show and Tell Language. The type system is polymorphic and includes several primitive types, constructs for defining tuples and unions of types, type variables and a method for naming types. Figure 27 shows the definition of the type 'stack of T: $\text{void} \cup (\text{T} \times \text{stack of T})$.'

Functions are provided to construct records, to select elements of records and to tag and filter union types. A multipipe represents n arcs between functions, where n is

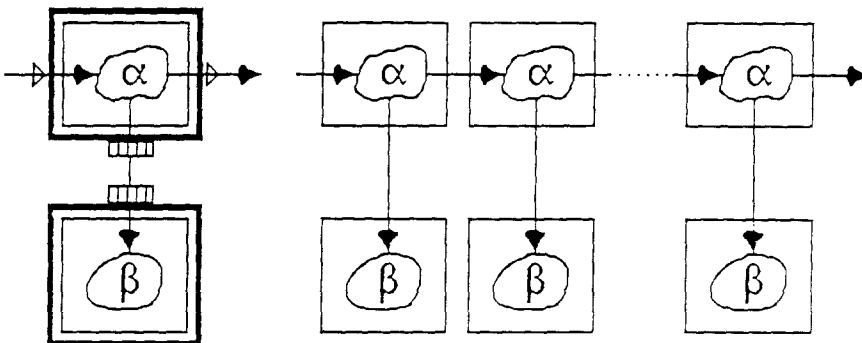


Figure 26. Parallel iteration in Show and Tell [6]

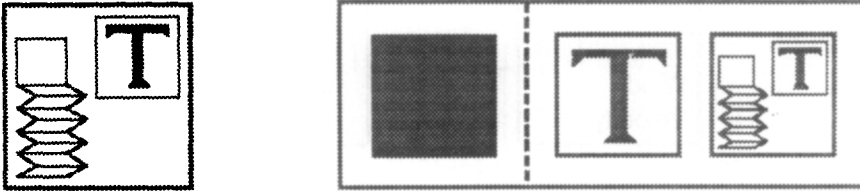


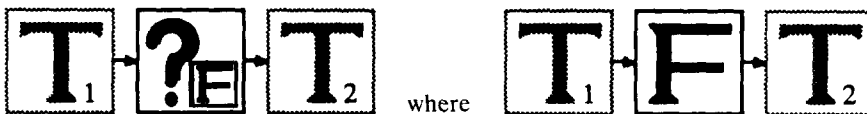
Figure 27. ESTL type declaration of 'stack of T' [10] © 1990 IEEE

variable. Multipipes are useful for carrying data between functions with varying numbers of arguments and results.

A function slot is a parameter to a higher-order function. A lower-order function is placed into the slot and is applied to data by the higher-order function. (This is similar to the function slot concept in Hils [11], but was developed independently.) A function slot's visual appearance is a parameter box in the icon for a function's name. The box contains a slot variable, which can contain a lower-order function. Figure 28 shows a higher-order function with a function slot containing the slot variable *F*.

3.7.2. DataVis

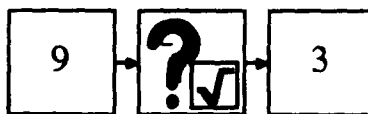
DataVis [11] is a language for the visualization of scientific data [30]. A central task of scientific visualization is converting scientists' numeric data into graphics so that scientists can better understand and analyse the data. DataVis is geared toward this aspect of scientific visualization and is intended to be used by scientists. Most scientists are not computer science or visualization experts who enjoy building basic software tools for visualization. These scientists usually prefer to concentrate on their field of science rather than on programming to develop software tools.



(a) Name



(b) Definition



(c) Use

Figure 28. ESTL higher-order function with a function slot [10]

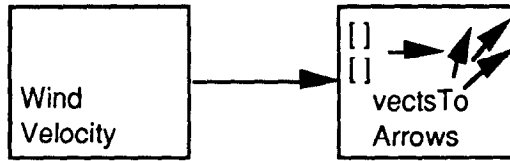


Figure 29. An example call to the DataVis library program `vectsToArrows` [11]

DataVis is in the same family of languages as Show and Tell. DataVis will provide a library of finished programs for visualization. Scientists may run any of these programs by connecting the program's icon to a data icon. Besides finished programs, DataVis will provide scientists with a set of 'building-block' functions which they may use to construct new programs and to modify and extend the finished programs.

Existing visual languages for scientific visualization are visual only at the top level. That is, the top-level presentation of functions is visual, but when one examines the internal workings of one of the functions (which internals can be thought of as a 'lower level'), the function is presented in a textual language. In contrast to such languages, DataVis will provide multiple visual levels of functions. Thus, when one looks at various levels of functions, DataVis will provide a consistently visual interface.

DataVis provides for higher-order functions with a function slot concept which is similar to, but was developed independently from, the function slot ideas in Najork and Golin [10].

`VectsToArrows` is a predefined DataVis program to transform a 2-D vector field into an arrow field display. An arrow field display has an arrow placed at each point in the vector field, where the direction of each arrow is the direction of the field at that point and where the length of each arrow is scaled by the magnitude of the field at that point.

Figure 29 shows one possible call to `vectsToArrows`. `vectsToArrows` takes one input parameter (a 2-D field, or collection, of vectors), creates an arrow field and displays it. When the scientist using DataVis decides that he or she wants an arrow field display for the data in the wind velocity vector field, he or she simply links the two icons (data and program) together. Since `vectsToArrows` is a predefined library program, the scientist does not have to create `vectsToArrows`, but may simply use it. Once the scientist executes the program, the arrow field will be displayed on the screen.

Figure 30 is the `vectsToArrows` program. The `2DVectorField` box is an input

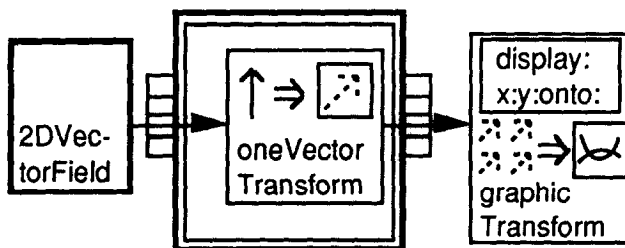


Figure 30. The DataVis `vectsToArrows` program [11]

parameter to `vectsToArrows`, and receives whatever data is flowing on the link connected to the `vectsToArrows` icon (Figure 29).

`VectsToArrows` contains two `DataVis` functions: `oneVectorTransform` and `graphicTransform`. `OneVectorTransform` is inside an iteration box which has a pair of parallel ports. The vector field is fed into the iteration box through the parallel port on the left of the iteration box. This parallel port disaggregates the vector field into its individual vectors. Each vector is fed to `oneVectorTransform`, which transforms the vector into a small bitmap containing an arrow of the proper orientation and length. The parallel port on the right side of the iteration box collects up all of the small bitmaps and forms them into a 2-D field of bitmaps when the iteration is finished. The `graphicTransform` function then takes the field of bitmaps, transforms the field into a graphic display (a single large bitmap), and displays the graphic display on the screen. Both `oneVector Transform` and `graphicTransform` have additional visual lower levels which are not shown here.

`GraphicTransform` is a higher-order function. The lower-order function, `display:x:y:onto:`, is in the function slot at the top of `graphicTransform`. `GraphicTransform` or one of its callees will eventually apply `display:x:y:onto:` to certain data.

3.8. Design Alternatives Table

Table 1 presents a summary of data flow visual programming languages and of their use of various design alternatives. A few notes may clarify the table's information.

With regard to the 'box-line representation' design alternative, recent versions of HI-VISUAL use juxtaposition of boxes to indicate that a function is being executed. *Viz* and Visual Toolset can use both lines and juxtaposition of function icons to show data transport between functions.

The 'selector/distributor' design alternative includes languages which provide functions derived from the selector or distributor. The 'flow of data' alternative has values of uni-directional or bi-directional. The execution mode is demand-driven or data-driven.

Viz and the Visual Toolset can use both lines and juxtaposition of function icons to indicate data transport between functions. Cantata users can choose either execution mode and either liveness level two or three.

4. Characteristics and Strengths of Data Flow Visual Programming Languages

When one examines the existing and proposed data flow visual programming languages, some characteristics and strengths of these languages become apparent. (Wherever 'language' is used in this section or the next section, assume it means 'data flow visual programming language'.)

Data flow visual programming languages have been most successful when they have a narrow, fairly specialized application domain (e.g. LabVIEW, whose application domain is collection and analysis of data from laboratory instruments), or when they are intended for use by non-programmers or novice programmers. Two languages in the second category are Show and Tell, which is a general-purpose

programming language, but is intended for school children, and LabVIEW, many of whose users had not programmed before using LabVIEW [7].

Due to their underlying data flow computational model, data flow visual programming languages work best when their application domain centers on data manipulation. Thus, languages with such application domains as image processing (HI-VISUAL, VIVA, Cantata), computer graphics (ConMan) and visualization of scientific data (DataVis, VIPEX) work well. However, data flow visual programming languages also work acceptably for application domains not centered on data. For example, Hookup, whose domain is music, works acceptably well.

The languages just mentioned (HI-VISUAL, ConMan, etc.) all use visual data. A strength of such languages is that their users are already accustomed to working with visual data, and may therefore be more willing than other users to accept a visual representation for their programs, too.

Data flow visual programming languages are a good choice to deal with the transformation of data (for example, in scientific visualization). Users of the language can watch as data flows from filter function to filter function, being transformed as it goes. The box-line visual representation which is typical of data flow visual languages easily allows the insertion of viewing monitors at various points to show the data to the user.

A data flow visual programming language may be considered *powerful* if it provides the necessary programming constructs and predefined functions so that a user of the language can tackle large, complex problems in the language's application domain. The following features are *necessary* for a data flow visual programming language to be powerful: iteration, procedural abstraction and a large and diverse library of predefined functions for use as building blocks in constructing new programs. Iteration makes programs more compact and expands the range of problems which a user of the language can address. Procedural abstraction is important to hide unnecessary details and to save screen space. An extensive library of predefined functions which are defined properly can reduce the time needed to create a program, and can help avoid recreating an existing function. Whether these features are *sufficient* to make a data flow visual language powerful is an open question.

Parameters for functions and higher-order functions are also desirable for making languages more powerful. They also, however, add to the complexity of the language.

5. Unsolved Problems in the Design of Data Flow Visual Programming Languages

As well as strengths, data flow visual programming languages have some weaknesses, and there are a number of possible areas for future research.

Like all visual programming languages, data flow visual programming languages take up a great deal of screen space. This difficulty can be alleviated by procedural abstraction, calculator boxes (for entering mathematical formulas textually), and the use, not just of graphical icons, but also of text, for names of variables, data objects, functions, classes and instances. Research aimed at finding new ways to address the screen space problem would be useful.

Often a language designer finds it necessary to add control flow constructs to data flow visual languages: iteration constructs based on control flow, sequential execution

constructs and **CASE** statements. The associated research topic is deciding which control flow constructs, if any, should be in a visual data flow language. The application domain and intended users of the language are important considerations in this decision.

The preceding section mentioned that a language must have a large library of predefined functions in order to be powerful. Research is needed to determine what the appropriate high-level library functions are for application domains in which data flow visual languages have not been used extensively over a number of years (for example, scientific visualization).

An area which can be investigated more is the implementation of data flow visual languages on parallel-processor architectures and on distributed systems. Such an implementation would be particularly useful for a language with a computation-intensive application domain: for example, image processing or some areas of scientific visualization. *apE* [17], while not a programming language, now allows a user to designate various function icons to be executed on remote machines. Presumably the icons so chosen require heavy computation for which the remote machines are better suited than the host.

Creation of languages which are applicable to many application domains has not been widely explored. Also, making data flow visual programming languages which are intended for use by professional programmers, not just novice programmers or non-programmers, is an interesting challenge.

While *Cantata* [12] allows the user to choose between two liveness levels (two and three) within a single language, no data flow visual programming language offers a choice of all three possible levels (two, three or four). (A visual programming language will never be at level one.)

A virtual reality approach to the user interface of a data flow language may be useful. For example, one could edit a visual data flow program by using a data glove [42] to grab and move an icon shown to a user via a head-mounted display. The icon would appear to be in three-space in front of the user. As another example, *CUBE* [43] is a visual language which will use a virtual reality programming environment.

Finally, finding new appropriate application domains for data flow visual languages is an area worthy of investigation. Some possible application domains include multimedia systems, hypertext and hypermedia and medical imaging. With regard to multimedia systems, data flow visual languages could be used to edit audio and video data.

6. Conclusion

A number of successful data flow visual programming languages exist or have been proposed. Data flow visual programming languages have the potential to expand the appeal of visual programming by applying visual programming to new application domains (thus introducing visual programming to new users associated with those domains), and by providing a receptive environment for increased display of data at various points within visual programs.

Acknowledgements

I thank Ralph Johnson for his support of this work, and Eric Golin, Donald Hearn, Marc Najork, Greg Rogers and Robert Sum for helpful comments.

References

1. J. B. Dennis (1975) First version of a data flow procedure language. Technical report MIT/LCS/TM-61. Laboratory for Computer science, MIT.
2. T. Agerwalak & Arvind (1982) Data flow systems: guest editors' introduction. *IEEE Computer* 15, 10-13.
3. A. L. Davis & R. M. Keller (1982) Data flow program graphs. *IEEE Computer* 15, 175-182.
4. M. Hirakawa, N. Monden, I. Yoshimoto, M. Tanaka & T. Ichikawa (1986) HI-VISUAL: A language supporting visual interaction in programming. In: *Visual Languages* (S.-K. Chang, T. Ichikawa, & P. A. Ligomenides, eds) Plenum Press, New York.
5. M. Hirakawa, S. Iwata, Y. Tahara, M. Tanaka & T. Ichikawa (1988) A Framework for construction of icon systems. *IEEE Workshop on Visual Languages*. Pittsburgh, Pennsylvania, 10-12 October, pp. 45-51.
6. T. D. Kimura, J. W. Choi, & J. M. Mack (1986) A visual language for keyboardless programming. Technical report WUCS-86-6. Department of Computer Science, Washington University, St Louis, Missouri 63130.
7. M. Santori (1990) An instrument that isn't really. *IEEE Spectrum* 27, 1990 36-39.
8. D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph & K. Doyle (1988) Fabrik: a visual programming environment. In: *Proceedings ACM OOPSLA '88*. September, pp. 176-190.
9. F. Ludolph, Y.-Y. Chow, D. Ingalls, S. Wallace & K. Doyle (1989) The Fabrik programming environment. *IEEE Workshop on Visual Languages*. Pittsburgh, Pennsylvania, October 10-12, pp. 222-230.
10. M. A. Najork & E. Golin. (1990) Enhancing show-and-tell with a polymorphic type system and higher-order functions. In: *IEEE Workshop on Visual Languages*. Skokie, Illinois, 4-6 October, pp. 215-220.
11. D. D. Hils (1991) DataVis: a visual programming language for scientific visualization. In: *Proceedings 1991 ACM Computer Science Conference*. San Antonio, Texas, 5-7 March, pp. 439-448.
12. J. Rasure, D. Argiro, T. Sauer & C. Williams (1990) A visual language and software development environment for image processing. *International Journal of Imaging Systems and Technology* 2, 183-199.
13. C. S. Williams & J. R. Rasure (1990) A visual language for image processing. In: *IEEE Workshop on Visual Languages*. Skokie, Illinois 4-6 October, pp. 86-91.
14. J. Rasure & C. S. Williams (1991) An integrated data flow language and software development environment. *Journal of Visual Languages and Computing* 2, 217-246.
15. C. Upson, T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz & A. van Dam (1989) The Application Visualization System: a computational environment for scientific visualization. *IEEE Computer Graphics and Applications* 9, 30-42.
16. M. VandeWettering (1990) The Application Visualization System-AVS 2.0. *Pixel* 1, 30-33.
17. D. S. Dyer (1990) A dataflow toolkit for visualization. *IEEE Computer Graphics and Applications*. 10, 60-69.
18. S. L. Tanimoto (1990) VIVA: a visual language for image processing. *Journal of Visual Languages and Computing* 1, 127-139.
19. E. P. Glinert & S. L. Tanimoto (1984) Pict: an interactive graphical programming environment. *IEEE Computer* 11, 7-25.
20. M. Hirakawa, S. Iwata, I. Yoshimoto, M. Tanaka & T. Ichikawa (1987) HI-VISUAL iconic programming. In: *IEEE Workshop on Visual Languages*. Linkoping, Sweden, 19-21 August, pp. 305-314.
21. D. Levitt (1986) Hookup: An iconic, real-time data-flow language for entertainment. Unpublished technical note. MIT Media Lab. Available from Hip Software Corporation, 117 Harvard St No. 3, Cambridge, Massachusetts 01239.
22. Claris Corporation (1988) *MacDraw II* Claris Corp., Mountain View, California.
23. D. N. Smith (1988) Visual programming in the interface construction set. In: *IEEE Workshop on Visual Languages*. Pittsburgh, Pennsylvania, 10-12 October, pp. 109-120.
24. I. Yoshimoto, N. Monden, M. Hirakawa, M. Tanaka & T. Ichikawa (1986) Interactive

- iconic programming facility in HI-VISUAL. In: *IEEE Computer Society Workshop on Visual Languages*. June, pp. 34–41.
25. M. Hirakawa, M. Yoshimi, M. Tanaka & T. Ichikawa (1989) A generic model for constructing visual programming systems. In: *IEEE Workshop on Visual Languages*. Rome, Italy, 4–6 October, pp. 124–129.
 26. Hirakawa, Masahito, Yoshimi, Makoto & Ichikawa, Tadao (1990) A universal language system for visual programming. In: *IEEE Workshop on Visual Languages*. Skokie, Illinois, 4–6 October, pp. 156–161.
 27. S. L. Tanimoto (1990) Towards a theory of progressive operators for live visual programming environments. In: *IEEE Workshop on Visual Languages*. Skokie, Illinois, 4–6 October, pp. 80–85.
 28. V. Haarslev & R. Moller (1988) Visualization of experimental systems. In: *IEEE Workshop on Visual Languages*. Pittsburgh, Pennsylvania, 10–12 October, pp. 175–182.
 29. V. Haarslev & R. Moller (1990) VIPEX: visual programming of experimental systems. In: *Visual Languages and Visual Programming* (S.-K. Chang, ed.) Plenum Publishing Corporation, New York.
 30. B. H. McCormick, T. A. DeFanti & M. D. Brown (eds) (1987) Visualization in Scientific Computing. *ACM Computer Graphics* 21, 1–14.
 31. National Instruments Corporation (1987) LabVIEW: a demonstration. National Instruments Corp., 12109 Technology Blvd., Austin, Texas 78727-6204.
 32. G. M. Vose & G. Williams (1986) LabVIEW: laboratory virtual instrument engineering workbench. *Byte* 11, 84–92.
 33. P. E. Haeberli (1988) ConMan: a visual programming language for interactive graphics. In: *Proceedings ACM SIGGRAPH '88*. Printed as *ACM Computer Graphics* 22 August, 103–111.
 34. C. M. Holt (1990) Viz: a visual language based on functions. In: *IEEE Workshop on Visual Languages*. Skokie, Illinois, 4–6 October, pp. 221–226.
 35. J. A. Borges (1990) Multiparadigm visual programming languages. Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign.
 36. J. A. Borges & R. E. Johnson (1990) Multiparadigm visual programming languages. In: *IEEE Workshop on Visual Programming Languages*. Skokie, Illinois, 4–6 October, pp. 233–240.
 37. S. Matwin & T. Pietrzykowski (1985) PROGRAPH: A preliminary report. *Computer Languages* 10, 91–126.
 38. P. T. Cox, F. R. Giles & T. Pietrzykowski (1989) PROGRAPH: A step towards liberating programming from textual conditioning. In: *IEEE Workshop on Visual Languages*. Rome, Italy, 4–6 October, pp. 150–156.
 39. T. D. Kimura (1986) Determinacy of hierarchical dataflow model: a computation model for visual programming. Technical report WUCS-86-5. Department of Computer Science, Washington University, St Louis, Missouri 63130.
 40. T. D. Kimura & P. McLain (1986) Show and Tell user's manual. Technical report WUCS-86-4. Department of Computer Science, Washington University, St Louis, Missouri 63130.
 41. T. D. Kimura (1986) Show and Tell sample programs. Technical report. Department of Computer Science Washington University, St Louis, Missouri 63130.
 42. J. D. Foley (1987) Interfaces for advanced computing. *Scientific American* 257, 127–135.
 43. M. A. Najork & S. M. Kaplan (1991) The CUBE language. In: *IEEE Workshop on Visual Languages*. Kobe, Japan, 8–11 October, pp. 218–224.