A VISUAL LANGUAGE FOR PARALLEL PROCESSING

by

Peter Klier

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1987

Signature of Author _____
Department of Electrical Engineering and Computer Science
June 1, 1987

Certified by _____
Ronald MacNeil
Thesis Supervisor

Accepted by _____
Leonard A. Gould
Professor of Electrical Engineering and Computer Science

Table of Contents

A VISUAL LANGUAGE FOR PARALLEL PROCESSING

by

Peter Klier

Submitted to the Department of Electrical Engineering and Computer Science on June 1, 1987 in partial fulfillment of the requirements for the degree of Bachelor of Science.

Abstract

A graphic environment for a visual computer language has been designed, and a graphic editor and interpreter for constructing and experimenting with this language has been implemented. This language is based on many previous implementations of dataflow languages and neural networks. It embodies within it many concepts of computer science such as procedural and data abstraction, and includes features such as menu-driven operation and high-quality color graphics. The visual language implemented is designed with parallel processing computers as the target machine.

Thesis Supervisor: Ronald MacNeil
Title: Principal Research Associate, Visible Language Workshop

Dedication

To Mom, Dad, John, Nancy, Babichka Simer, Babichka Klier, Martin, Ota Klier, Helen, Jitka, and Ota Jr.

1 Introduction

The newly developed connection machine is a computer like no other that came before it. Consisting of over 65,000 parallel processors, it holds within it the promise of unlocking yet untapped wellsprings of computer power. As with any powerful new invention, methods must be developed to harness this power to its fullest extent. To this day, the programming languages that are used to program the connection machine and other parallel machines are based on a serial rather than parallel model of computation. Furthermore, a computer language consisting of written text is difficult to convert into a semantics of parallel computation. Obviously a new means of expression for this new machine must be found, one that is easy to use, and yet well suited to the target machine.

A language appropriate for parallel processing is awkward if it is represented in a sequential form. A user can better program in and understand a parallel program if he is allowed to use the non-sequential faculties of his mind , i.e. the visual system. Therefore, an entirely visual language for parallel processing is a desirable thing to have.

The language that I propose for use on the Connection machine is one that consists of hooking up procedural "objects" on a graphics screen, and also of abstracting such "objects" into procedures. This language can be made as powerful as C, Lisp, or any other conventional language. When the user hooks up these objects, he tells the computer, via the syntax of the visual language , the data dependencies involved, allowing the compiler or interpreter to make accurate and efficient decisions pertaining to the implementation. This feature makes the language portable, not only across parallel computers, but also from parallel to serial computers and back again.

2 Background

The notion of a visual language for parallel processing has its origins in many different previous ideas. One of the most important ideas involved is that of machine independence. Languages, such as CLU, C, and Common Lisp, all strive toward this goal, trading off machine independence against efficiency and versatility. Another important idea is the notion of abstraction. There are various kinds of abstraction, including procedural, control, and data abstraction. Yet another idea that contributed is the prevailing trend for computer programs to deal with their user in a friendly manner, and especially for this user interface to be graphic, as is found in menu systems and in programs such as Lotus 1-2-3. The general idea that a programming language should be as simple and uniform as possible had a significant role in the design of the language. And finally, the basic nature of the language was, in some sense, synthesized from the natures of existing paradigms, such as neural networks, which are able to implement many different algorithms and solve NP=-complete problems approximately, the forward and backward chaining properties of expert systems, and the hierarchy of frame-based knowledge systems.

2.1 Machine Independence

Many computer languages are designed to be machine independent, that is, it is desirable to have the property that the same program performs the same task when it is run on different machines. Such machine independence is a property of the language. For instance, Clu is almost completely machine independent. The key to machine

independence is to build your programming language on a sufficiently general machine model so that it encompasses all possible computers. Since ordinary languages are built on a step-by-step model of computation, it is clear that they are not entirely suitable for parallel processing. Therefore we must build our new language on a parallel computation model, one in which all timing constraints are explicitly given by the programmer. For this reason, it was chosen to have a data-flow model of computation, where each data value must be a valid result of a previous computation. In addition, it is desirable to embody this feature into the fundamental nature of the language. Therefore, we want a program to look something like a data-precedence graph.

2.2 abstraction

In the languages C, CLU, and Common Lisp, there exist mechanisms for procedural, data, and control abstraction. Procedural abstraction hides the details of how a computation is performed from the user of that computation; Data abstraction hides the representation of a data object from its user, and control abstraction hides the details of control decision-making from the user. All these forms of abstraction are very desirable. However, in a parallel computation environment, the distinction between procedures and data blurs,. Both are some sort of object with a state, inputs and outputs, and relationships among such states. In serial processing, the distinction between procedure and data comes from the fact that we do not "copy" a procedure, since we can only use one procedure at a time. However, in parallel processing, each invocation of a procedure is treated as a separate object. Thus, we can represent both an abstract procedure and an abstract data object as an instance of a particular type, whose

representation is hidden from the user. Once we dot his, control abstraction comes for free, because under our data-precedence constraints, we can hook up such an "object" into a loop and have it compute next states in sequence.

2.3 User Friendliness

We also want our programming language to be user friendly, easy to learn, and easy to use. Above all, it must be as simple as possible. Computer programs such as Lotus 1-2-3 and "Hookup" allow the user to actually program the computer without really thinking of it as 'programming'. In parallel processing, where programs can become extremely complex, simplicity is very important. Thus, having a visual language with only one basic method of hooking up objects and one method of abstraction, using 'primitive' types to handle what is handled now by 'constructs' in today's programming languages.

2.4 Previous Ideas

There are many areas of research which contain ideas relevant to designing the visual language. For example, expert systems employed in artificial intelligence use forward and backward chaining similar to this language's data dependency mechanisms. Neural networks, which have been implemented on microchips, are especially suitable for this language, since they themselves have imprecise timing constraints. Many applications of neural networks have already been realized: Boltzmann machines, which can simulate probabilistic quantum-mechanical processes such as metal annealing and neural networks that solve the Traveling Salesman problem. the

language is also similar to circuit simulators and block diagrams used in engineering of all kinds.

3 Basic Operation of the Visual Language

The language I propose is a simple one. Unlike most computer languages, programming in this language involves manipulating visual symbols rather than writing text. It is designed this way to incorporate parallel processing directly in a description of a computer program, rather than resorting to an awkward circumlocution. It embodies within it many concepts of computer science such as data and procedural abstraction. This language is best suited for a MIMD(multiple instruction multiple data) machine, but it is easily adapted to a SIMD(single instruction, multiple data) machine with a slight sacrifice in efficiency.

The basic idea of this language is to write a visual program as a connected network of "objects", represented on a computer screen as a two-dimensional visual symbol. Each "object" has a certain number of (zero or more) "arguments".Each "argument" "pipe" or "wire" to one or more "arguments" of another "object". Data flows through these pipes or wires and supplies each object with values that it needs to perform a computation. After an object has performed a computation, it modifies some of its arguments, and thus sends more data along more pipes. The data in these pipes, if attached to other objects via the objects' arguments, initiates a new computation in these other objects. This process continues indefinitely.

3.1 Data Flow and Parallelism

Every object in the system performs its computation simultaneously with every other object. The only timing constraints in the system occur as a result of the limited availability of valid data on the pipes. Each pipe initially holds no value. In order to perform a computation, an object must have data available to it. Thus, an object will only begin a computation when enough of its arguments have become defined, and only then will it constrain other of its arguments, causing more pipes to become defined, causing other objects to perform their computations.

We can think of writing(drawing) a program in this manner as setting up a system of pipes and valves. We can think of data as water flowing through pipes, and objects as special-purpose valves. Initially, most of the pipes are filled with air(undefined). Then we turn on the water by constraining a few pipes to constants. The water begins to flow in these first pipes, and activates a few of the special valves(objects). The valves thus activated then pour water into other pipes. This process continues indefinitely until most of the pipes are full of water.

If we want to perform a computation with our pipes-and-valves system, we can, for example, use water temperature as a means for carrying information. Our special valves would add, subtract, multiply, divide or do anything at all with water temperatures. However, an ordinary valve cannot perform a computation based on air temperature. Thus it cannot begin pouring water out until water pours into it, just as an object cannot perform a computation until sufficiently many of its arguments are constrained.

4 Types and Abstraction

Each object in a program has a given type. There can be many objects of the same type in different places, connected to other objects of the same or a different type, much as one can call the same procedure from different places in the same program. The type of an object determines the number of its arguments and also determines what computation it performs. Thus a type, in our pipes and valves analogy, is like a model of a valve. Each valve model has the same relationship between input and output temperatures. But two valves of the same model, since they are connected to different pipes, can pour water of different temperatures at their outputs.

4.1 Primitive Types

Performing a computation through an object of a given type is the only method that the programmer may use to control the data flow and the timing of the program, and thus it is necessary that objects perform control operations as well as simple arithmetic . There are two basic classes of types, although they are both used in the same way: user-defined types and primitive types. Primitive types are always available to the user , and perform standard computations such as addition, subtraction, multiplication, and division, as well as control operations such as detecting empty pipes, if-then, storage, and other operations which allow the programmer to control the data flow. These primitive types have a standard appearance on the user screen as graphic icons.

4.1.1 Arithmetic Primitive Types

Arithmetic primitive types generally have two or more arguments. One of these arguments will act as an output, with the rest acting as inputs. In general, an arithmetic primitive type will not constrain its output until all its inputs have been defined, thus ensuring a correct answer. Primitive arithmetic types include addition, subtraction, multiplication, division; logical or, and, not, and exclusive or.

4.1.2 Control Primitive Types

Control primitive types are designed to allow the user greater control over the flow of data in a program. Their behavior depends on the specific type. Two of the control primitives are the "if" and the "defined" primitives. The "if" primitive has two inputs and one output. One input is designated as the 'control' and the other as the 'data'. The "if" primitive behaves in a manner such that the output is constrained to the same value as the 'data' input only if the 'control' input is constrained and has the value 'true'. The 'defined' primitive has one input and one output, and always constrains its output. This output is constrained to the value "true" if the input is defined(constrained), and "false" if the input is undefined(the pipe contains air).

4.2 User Defined Types

In ordinary programming languages, there are often mechanisms for procedural and data abstraction. A crucial part of the visual language for parallel processing is the ability to create user-defined types, which perform a certain user-defined computation. For example, one may wish to calculate the roots of a quadratic equation, given the initial coefficients, for two different quadratic equations, and furthermore do this in parallel. To do this, one would create a user-defined type, just as one would define a procedure in an ordinary programming language.

4.2.1 Creating a User Defined Type

A user defined type consists basically of three components: an appearance, a set of arguments, and an implementation, which is a connected network of objects, just as an ordinary program is. To define a new type, one simply writes(draws) a program, defines the number of arguments and where they will be placed on the appearance of the type, and connects these arguments via pipes to the arguments of objects in the program. Then one creates the desired visual appearance of the user defined type, including places for the arguments. Finally, one tells the visual editor that one wishes to define the current program, arguments, and appearance as a new type, and thus the type is installed, ready to be used as if it were a primitive type.

Thus, this programming language provides a simple and uniform mechanism for programming parallel processing. Provided with the appropriate primitive types, and the proper means of combination through pipes, the programmer in this language has as much power as he would have in any programming language, and more.

5 Implementation

The visual language for parallel processing is implemented by building the appropriate data structures that represent visual programs, both their appearance and the underlying connections between visual program objects. While most programming languages simply rely on a text editor as sufficient to create programs, the visual language requires a graphic editor. In order to simplify matters, the graphic editor will build the underlying data structure of a visual program as it builds the visual representation on the graphics screen. This is done because the visual and underlying representations of the visual program are closely related.

The graphic editor is written primarily in common Lisp on a Hewlett-Packard Bobcat computer. Common Lisp was chosen as the primary language of implementation because of its support for data structures and abstraction. The Bobcat computer was chosen because of its extensive graphic capability. For speed, some underlying code was written in C.

5.1 The Basic Representation of Visual Objects

For various reasons including space efficiency and ease of translation, scaling, and duplication, and compatibility with the underlying representation, a visual object on the screen is not represented as a simple bitmap, but as a recursive mathematical lattice structure. Each icon is represented as either a primitive visual object for which drawing, scaling, and translation methods are known , or as a composite icon object which includes scaling and translation information, as well as a set of 'sub-icons', which are themselves icons. Two separate icons can, of course, contain the same 'sub-icon'. These two separate icons can, in turn, be sub-icons of the same top-level icon, hence a lattice structure. If we had required that no two icons can contain the same sub-icon, then this structure would be represented as a tree. Such a structure facilitates, for example, drawing two AND gate icons in different places. We merely create two separate icons with different scaling and translation information which both contain as sub-icons the AND gate icon. This saves computer memory because we need not duplicate the AND gate icon data structure.

Once we have such a lattice structure of icons, we need only to translate it into visual

objects on the screen. This is done by means of a translation/scaling algorithm[1] which uses as input a translation, a scale, and an icon and produces as output another translation and scale. This new translation and scale is used as a basis for applying the algorithm to the next sub-level of icons in the lattice if such a sub-level exists. If no such sub-level exists, then the object contained in the icon must be a "primitive" type of object with a known drawing method relative to a given translation and scale.[2]

5.1.1 Implementation in Common Lisp

The obvious choice for the implementation of icons in common Lisp is to use the data abstraction facility already provided of common lisp Objects. Using common lisp objects greatly facilitates the implementation of icons for various reasons.[3] One of these reasons is the fact that we can apply a "method" to such an object without knowing the object's instance type. This allows us to perform operations such as redrawing a primitive icon just by giving it the scale and translation factors to operate upon. In addition, this gives us the facility to continuously add primitive icon instance types and provide a richer visual repertoire. Also, this allows us to send a message to an instance object such as "convert yourself to a list", without worrying about its instance type, and thus greatly facilitates the coding of these objects onto files.

An icon consists of several essential parts. These are:

1. A translation factor (X and Y offset)

---

[1]This algorithm is called 'rescale', found in /net/leo/lisp/users/pklier/zicon2.l

[2]This entire paradigm is implemented by the algorithm 'repaint-icon', found in /net/leo/lisp/users/pklier/zicon2.l

[3]The code is found in /net/leo/lisp/users/pklier/zicon2.l as a 'define-type' of "icon"

2. A scale factor (generally 1.0)

3. A boolean value "primitive" which indicates whether the icon contains sub-icons or not

4. Either

    a. A list of sub-icons or

    b. A primitive,drawable object (currently an arc, regular n-gon, irregular polygon,rectangle, text, or a special object known as a "color triangle")

Each primitive, drawable object (arc,regular n-gon,irregular polygon,rectangle,text,or color triangle) has no scale or translation information associated with it, but contains special information, the nature of which is particular to the instance type in question. For instance, a regular n-gon contains the number of vertices and the initial angle of the n-gon. When drawn, the n-gon must be given an x and y offset and a scale, which will then become the center and radius of the regular n-gon. Similarly, an arc contains an initial angle and a final angle, with its center and radius determined by the x and y offset and scale given to it by the algorithm[4] mentioned above.

5.2 The Basic Control of the Graphic Editor

Virtually all control of the graphic editor is done via menus that appear at various places on the screen. In general, the user moves the mouse to some menu item, presses a button, and invokes some editor function which affects the program structure by creating an icon, moving an icon, performing i/o, and so forth. In addition, it is necessary to have such areas of the screen be affectable dynamically, so that they can

---

[4] 'rescale'

be moved, removed, and so forth. We want the user to be able to pick different icons as they appear on the screen, connect them, pick colors, pick icon parameters, exit the editor, drag icons, and so forth. For simplicity's sake, and to save time in debugging, and for code readability and understandability, it is best to have all editor functions handled by a single mechanism. In addition, it is desirable for this mechanism to be high-speed and non-linear in the total number of active menu areas.

5.2.1 The Region Mechanism

The mechanism employed involves the use of a recursive tree structure known as a "region". It is a very simple structure(implemented by common objects). Basically, each region is a rectangular area of the screen which contains zero or more subregions(there is no limit). Each subregion within a region is a rectangular area that is wholly contained within the rectangular area of the parent region. Therefore, we can find the smallest region containing the coordinates of the mouse in O(log n) time in the total number of regions by searching down the region tree. We need only search one subregion at every depth level, since it is guaranteed that all the subregions of a particular region are wholly contained within their parent region. Such a tree structure also facilitates operations such as treating menus, which themselves may contain many menu items as subregions, as whole objects which can be manipulated. Note that the region structure, unlike the icon structure, is a strict tree, rather than a lattice.

Each region, in addition to its structure information, contains other information which is pertinent to control over the functions of the graphic editor. These include

1. Possibly an icon

2. A function which is invoked when the mouse enters the region, typically making the region brighter

3. A function which is invoked when the mouse exits the region, typically making the region darker

4. A function which is invoked repeatedly while the mouse is in the region, typically one that only does something when a button is pressed.

Typically, the mouse enters a region, it brightens, and then the user presses a button. The region's function, being repeatedly called, senses this button push and invokes some other function, which may, for example, exit the editor , ask for text, ask for information, provide information, create an object, save the screen, and so forth. This function may, indeed, create a new region and put it up on the screen as a menu in order to get the information it needs.

5.2.2 Implementation in Common Lisp

Regions are implemented again as Common Lisp objects, primarily because of the support for data abstraction.[5] Each region contains

1. The x and y coordinates of the lower left corner

2. The x and y coordinates of the upper right corner

3. A list of subregions, possibly length 0

4. An entry function

5. An exit function

6. A continuously called function

---

[5]The code for regions is found in /net/leo/lisp/users/pklier/regiondef.l

7. An icon

8. Auxiliary information

When given an x and y coordinate (from the mouse, typically), we can find the smallest containing region[6] by starting with the root (the whole screen)[7] , and searching its subregions until we find a subregion that contains the x and y coordinates, repeating the same process on that region's subregions, and so forth, repeating until there exist no subregions of a particular region which contain the desired x and y coordinates(This, of course, includes the case where there are no subregions at all). In this case, the last region that contains the given x and y coordinates is the smallest containing region.

Thus, the entire program is controlled by one simple command loop. This command loop is responsible for getting the proper mouse coordinates and button values and invoking the proper function at the proper time.[8] This loop operates on the 'whole-screen' region tree, determining entries and exits to and from regions as well as which region the mouse is in. When the program is loaded, the 'whole-screen' data structure is set up, and thus all we need to do upon initialization is to repaint this region[9] and then invoke the command loop on it.

---

[6]method (region :smallest-containing)

[7]This is the variable "whole-screen"

[8]This code is the function 'start_up' found in /net/leo/lisp/users/pklier/editor.l

[9]function region-repaint found in /net/leo/lisp/users/pklier/editor.l

5.3 Creating an icon on the screen

Once the command loop is invoked and the appropriate menus are installed, we can then begin to create graphic icons on the screen. Using the region and menu mechanism to its fullest advantage, we include in our repertoire one menu item for each possible primitive kind of icon to draw, as well as one menu item for each element in the set which is the union of all possible attributes of each primitive drawable icon. For example, if we were only concerned with creating arcs and regular n-gons, we would have menu items for: arc, n-gon, center, radius, initial angle, final angle, and number of vertices. In addition, we allow the user some leverage in the method by which he chooses parameters for the creation of such objects. For example, the user may want to choose a center and an edgepoint, rather than a center and a radius, to define the limits of his graphic object. Furthermore, patterned after EZWin, we want to allow the user to choose the parameters to his object in any order, creating an object as soon as all the information needed to create such an object is available. As an additional feature, the program tells the user, by changing the color of the menu items, which parameters he still needs in order to create an object.

5.3.1 Implementation in Common Lisp

All these features are handled by one simple mechanism based on a data structure known as an and-or tree. Such a tree is easy to implement in common LISP, as we need only to write down its list structure.[10] Each primitive, drawable type has a different

[10]The code for this is found in /net/leo/lisp/users/pklier/oplist

and-or tree describing what kind of information it requires in order to be created. This and-or tree is augmented with numbers of features required at the leaf level.

There is also a generic and-or tree which describes the possible requirements for any basic icon element (n-gon,arc,text,polygon). For example, we wish to express the fact that in order to create a regular n-gon, we need

1. A center
2. A number of vertices
3. Either
    a. An edgepoint OR
    b. A radius

since we can calculate a radius from a center and an edgepoint. This notion is expressed in Lisp as

```
(and ("center" 1) ("vertices" 1) ("initangle" 1)
(or ("edgepoint" 1) ("radius" 1)))
```

Given a set of defined properties, we can determine what properties need yet to be fulfilled by a simple recursive algorithm. If a tree is an AND tree, then we need yet the union of what is required for all its subtrees. If a tree is an OR tree, then we need nothing if one of the subtrees requires nothing, otherwise we need the union of everything required by all the subtrees. Finally, if a tree is a leaf, we need the number of items required by that leaf minus the number of items present.

5.4 Choosing Colors

Color selection is done by a rather simple mechanism. Our goal is to be able to tell the computer an arbitrary vector in (red,green,blue) space. The method chosen to

implement this involves choosing the normalized vector along with the magnitude of this vector. Thus the user separately chooses the color and intensity. Furthermore, we need only be concerned with vectors in the first octant and, since the intensity of each color is limited, to a cube in RGB space.

Choosing a vector in a 3-dimensional space can be done from a 2-dimensional graphics screen if we have 2 separate areas determining the color, a one-dimensional control of intensity and a two-dimensional control for the normalized vector of the color mix. Since we only need normalized vectors in the first octant, it is easy to provide color control by displaying a slice of RGB space. This slice is naturally chosen to be the plane containing the points (0,0,1) (0,1,0), and (1,0,0) , cut off by the coordinate planes. This slice is an equilateral triangle with red, blue, and green vertices, gray in the middle, and with various color mixes in between. The equation of this plane is simply

```
R + G + B = 1.0
```

5.5 Anti-Aliased Fonts

In order to improve the aesthetic qualify of the language, which is, in fact, one of the goals of the language, it was desirable to implement the use of spatially anti-aliased fonts, which are very high quality visually. These anti-aliased fonts are encoded as bitmapped icons with two bits, or four distinct values, per pixel. It is necessary to map these values to actual colors before drawing an anti-aliased character. Code from Paul Chernoch, who implemented routines to write anti-aliased fonts of a fixed color against a fixed background, was modified so that an anti-aliased character of any given color

could be written anywhere on the screen, against any background, without overwriting this background, and in addition preserving the 'soft' nature of the colors intermediate between the foreground and the background, necessitating interpolation. All such calculations must be performed quickly and they must be performed "on the fly", since at any time we can ask for any foreground against any background. Ideally, this would be done on a picture-element by picture-element basis, but for efficiency reasons, it is done on a character-by-character basis, so that characters which cross a very harsh color boundary between one background and another may look funny.

Similar to the philosophy of a cache or a virtual memory, it is also safe to assume for our purposes that requests for the same foreground and background colors to be drawn for a particular character will probably occur many times in a row. When a request is made to draw a particular character against a particular background, a check is first made as to whether that particular (foreground,background) pair is available in the character's current bitmap. If not, an interpolation is made for intermediate colors between the foreground and background, the character's bitmap is re-colored according to these calculations, these colors are noted, and the character is displayed. Since demands for character printing may vary wildly, however, it is still essential that the interpolation of colors, the re-coloring of a font, and its display against a background be extremely fast and extremely efficient. By careful coding in C, this goal has been achieved.[11] With little modification, the code can be made to display characters against a background on a pixel-by-pixel basis, so that they do not look funny under any circumstances.

[11]This code is found in /net/leo/lisp/users/pklier/{fonts.c,typetec.c,colorchar.c} and other files associated with /net/leo/lisp/users/pklier/makefile The top-level routine is called "centered_disp_str".

5.6 The Tight Command Loop and Controlling Garbage Collection

In order for the mouse to control the editor effectively through the region mechanism already mentioned, it is necessary to continuously sample the mouse's (x,y) position. Since regions are implemented in Lisp, it is necessary to have these values continuously given to the Lisp system. This becomes a problem, however, since the interface between Lisp and the rest of the world(i.e. Starbase or C) is not particularly efficient. The problem lies with the fact that the particular implementation of Common Lisp we are using implements floating points number objects as pointers rather than as straight numbers, presumably to save memory. For example, we could have two variables, X and Y, which both have the value "2.3". They would then each contain a pointer to the same location in memory, the location where the value "2.3" is stored.

The problem occurs when we wish to pass a pointer to a floating point number out of the Lisp environment into the domain of ordinary C or assembled code with the intent of the external code modifying the value found at such a pointer.[12] The Lisp system, in order to preserve its integrity, must create a copy of the floating point value of the variable in question every time any external routine is called. For example, if both X and Y pointed to the same object "2.3", and we did not create such a copy when calling an external routine with a pointer to X, this external routine would also modify the value of Y, which is clearly in error. In a tight loop, millions of new floating-point copies may be made every second, causing frequent and long garbage collections. Such garbage collections may take up to 70 or 80 % of the total running time of the editor, something that is clearly unacceptable.

[12]Such a problem occurs in the Starbase routine "sample_locator"

The solution to this problem seems simple: Pass the pointers to the external routine before entering the tight loop, with the external code holding the values of these pointers. Then call another, parameterless, external routine, which simply modifies the values pointed to by the stored pointers. This solution works quite well, up until the time the system garbage collects. After a garbage collection, the pointer stored in the external code no longer necessarily points to the variable in question, and thus this data is inaccessible. Even worse, this pointer may point to a Lisp object of a different type, including a tagged data object. Modification of such a memory location could be disastrous to the Lisp system. Thus, it is necessary to detect garbage collection and re-initialize all the pointers stored in the external code after each garbage collection. In addition, such garbage collection detection must be done without causing the Lisp system to create new objects, because this detection must be done in the tight loop. But this seemingly impossible problem does, indeed, have a solution. In general, this solution consists of letting external routines do all the work. First, one tells the external routine to store a particular floating-point value, one which would never otherwise be generated,[13] in the location that it has remembered. Then the Lisp system finds the value of the variable in question, and passes this value to an external routine which compares this value with the value at the stored pointer. If these two values are the same, the external routine is still capable of successfully modifying the correct Lisp variable, and thus no pointer re-initialization takes place. If these two values are not the same, then this means that garbage collection has occurred and pointer re-initialization is to take place.

---

[13]This code is found as the procedure "start_up" in the file /net/leo/lisp/users/pklier/editor.l

5.7 Underlying Representation of Neurons

The underlying representation of a program in this language is an independent entity, the code for which was written by another student, Randy Sargent. Basically, objects are represented as structures, with properties including input signals, output signals, underlying representation objects, and internal and external connections. Thus, an entire program , in its distilled interpretable or compilable form, is a sort of directed graph.

Each object has a set of input signals and set of output signals. Each signal, in turn, contains information about whether its value is defined or undefined, and what that value is. In addition, each signal contains the objects that contain it, so that it can affect these objects when its value is changed.

There are two basic kinds of objects in the Common Lisp interpretation version of the language : primitive objects and abstract objects. When a signal changes that is an input to a primitive object, the Lisp function associated with that primitive object is evaluated, using the available defined signals as arguments to the Lisp function. The results of this Lisp evaluation are then used to define the values for some of the neuron's output signals. The Lisp function can choose to define or not define the output signals given a certain set of input signals, since it is written in a special format[14] . When an input signal to an abstract object changes, this object uses information about its internal connections to its defining object/signal graph and passes its own input signals to the input signals of its defining graph.

[14] see /net/jonesee/lisp/users/rsargent/defs.l

5.8 Interpretation of a Graphic Program

Presently an interpreter exists for the visual language described herein. This interpreter provides a graphic editor and an interface between graphic icons and graphs of objects and signals. After the user has finished drawing a program and he wishes to interpret this program, he invokes the interpreter, which creates an object/signal graph from the visual representation and begins to interpret it.

The interpretation of the object/signal graph is done by back and forward propagation of signal values. A global set of output signals of all objects is kept, and all output signals of all objects are "latched". The interpreter loops, and attempts to evaluate, by back propagation, each of the output signals kept in our global set. To avoid infinite loops when feedback in the graph is encountered, however, this back propagation is stopped after one level, which adds more signals that need to be evaluated to our global set of signals. The back propagation is advanced one level by calling the Lisp routine "clock". Thus, signals are evaluated in breadth-first order, avoiding infinite loops. These signals may be output signals, or they may be input signals of an object for which an output value was requested. We then keep looping, evaluating signals which need to be evaluated, until the user stops the interpretation.

When one of the signals that need to be evaluated acquires a defined value, perhaps by connection to a constant, forward propagation begins, and more input signals become defined. These defined input signals in turn cause Lisp evaluations, which define output signals. Thus, forward and back propagation are intermixed.

Input/output to this program is simple, as we only need to define primitive objects , with an implementation in Common Lisp, for this end. Thus, one of our objects is a "SHOW" object, which can be connected to any output, which will print on the screen the value of that particular output, every time said value changes.

6 Instruction Manual for the Graphic Editor

The Graphic Editor for the Visual Language is run from a Common Lisp environment on a Hewlett-Packard Bobcat computer. Currently this graphic editor can only be run from the NMODE text editor. The code for the Visual Language is found on "Leo", under /lisp/users/pklier. The main driving file for the load is "main.l", and the top-level routine for the editor is "start_up".

6.1 Getting Started

In order to run the visual language, it is necessary to set the environment variable "RFAPATH", start up an NMODE environment, and load in the appropriate code. Once in the NMODE environment, simply execute the form "(start_up)". To run the visual language, you must either log onto "Leo" as "pklier", in which case NMODE should be started and the code loaded automatically, or log onto some other machine and perform the following steps:

1. Set the environment variable RFAPATH to "/net/leo/lisp/users/pklier"
2. Netunam to "Leo" as "pklier"
3. Make sure you have set the environment variable RFAPATH to "/net/leo/lisp/users/pklier"

4. Start NMODE

5. Load the code by executing the Lisp form '(load "$RFAPATH/.nmoderc")'

6. Start the editor by executing the form "(start_up)"

Loading the code can take from 10-60 minutes, depending on the network traffic and the compilation status of the code. About 3 minutes after executing the form (load "$RFAPATH/.nmoderc"), the screen should begin to display the names of files which are being loaded. If NMODE tells you that it cannot find "$RFAPATH/.nmoderc", then you have either not NETUNAMed properly, which can be rectified by executing the Lisp form '(lan:netunam "/net/leo" "username:password")' , or you have not set the environment variable RFAPATH, which can only be rectified by exiting NMODE, setting the variable properly, and starting up another NMODE.

6.1.1 Using the Graphic Editor

In order to use the graphic visual language editor, which is an entirely screen menu driven system except for text entry, it is necessary to have a four-button mouse attached to the graphics tablet, affectionately known as a "rat". This mouse's buttons are numbered as follows:

```
      3
1             2
      4
```

In general, button 1(the left button) , is the "select" or "activate" button, button 4(the bottom button) is the "abort" button, and buttons 2 and 3 perform special functions. The function of a button is determined by which "region" the mouse is in. Often such

regions will light up when the mouse enters them, and darken when the mouse exits. Lighting up means "the mouse is inside me, press a button and I will do something."

6.1.2 Exiting the Graphic Editor

The most important thing you can know is now to exit the graphic editor. Just put the mouse inside the green region at the lower right hand corner of the screen, making sure this region lights up. Then press the top button(number 3). You can re-start the editor now by executing "(start_up)" or "(f)".

6.2 Creating a Graphic Icon

6.2.1 Simple Creation

In order to draw an abstract program, you will want to create the visual representation for each particular procedure. To do this, you will want to give the computer information about what the icon looks like. You can create an icon, and give it arguments, by selecting one of the items (rectangle, arc , text, polygon, n-gon, or argument) from the top menu, or one of the icons from the second-from-top menu.

To use the top menu, put the mouse into one of these areas, and the region around the primitive type should light up. Press button 1, the left button. Exit the region, which should remain lit up. You can change this selection by simply pressing button 1 in a different area of the top menu, or un-select all items in the top menu by pressing button 1 in the "clear" item of the right menu. Using the second-from-top menu is similar.

You can choose a color for your new object by pressing button 1, the left button of the rat, inside the color triangle in the lower right-hand corner of the screen. The rectangle above this area will show the current selected color. You can vary the intensity of this color by pressing the left button of the mouse, button 1, inside the blue bar on the left of the screen. When you do this, the little yellow square will jump to the mouse position and the color displayed by the rectangle above the color triangle will change.

Once you select an item from the top menu, some of the items in the right menu should turn red. An item in the right menu that is red indicates that the menu item is irrelevant to the type of object you wish to create. You can now give the primitive object the characteristics it needs. Once all the information necessary to create an object is present, the object is created. To give an object a characteristic, go to the right menu and press button 1 in the region appropriate for such a characteristic. The active menu item will remain lit until you give the computer the information it needs, or until you abort by pressing button 4. Each characteristic in the right menu will ask for information in the following manner:

1. Center
   Simply point the mouse at the desired center of the new graphic object, and press button 1. A small point should appear and the "center" item on the right menu should darken after a small mouse movement. You can press button 4 to abort if you do not wish to choose a center yet.

2. Radius
   Two bars will appear on the screen. The distance between these bars is the radius. The right bar follows the mouse. When the two bars are the desired distance apart, press button 1 and the distance between the bars will be registered as the radius. You can press button 4 to abort if you do not wish to choose a radius.

3. Edgepoint

Choosing an edgepoint is just like choosing a center. You can choose multiple edgepoints with button 1 or abort with button 4.

4. Initial Angle

   When the initial angle menu item is activated, two lines, like hands on a clock, will appear on the screen, and one will follow the mouse. They will appear at the right if no center has been chosen, or at the chosen center if such a center exists. The angle between these two lines will be chosen as the initial angle if you press button 1. No initial angle will be chosen if you abort by pressing button 4.

5. Angle

   Choosing the angle (of an arc,perhaps), is just like choosing the initial angle. Note that this parameter measures the total angle of an arc rather than the final angle of the arc.

6. Number of Vertices

   After choosing "vertices" on the right menu, a popup menu will appear near the top of the screen with numbers in it. Put the mouse in one of these areas, press button 1, and you will have chosen the number of vertices for your polygon or regular n-gon.

7. Text

   After choosing the "text" item, NMODE will prompt you to enter text in the NMODE window.

6.2.2 Dragging Created Icons Around

You can drag an object that you have created around by putting the mouse position on that object and pressing button 2, the right button. This object will then follow the mouse until you press button 1, at which point the object will stop following the mouse and appear at the new desired position.

6.2.3 Making Icons Look Pretty

To make your group of icons look pretty, invoke the "pretty" function from the right menu by pressing button 1. This will attempt to match points that are near each other

so that they are exactly at the same place, and to put things that are almost in line vertically or horizontally into exact alignment.

6.2.4 Grouping Icons, Installing a New Icon, Saving

When you want a group of icons to act as one object for the purpose of dragging or abstraction, select the menu item "group" on the right menu with button 1, the left button. To save this new icon to the icon menu, second from the top, first group it, and then invoke the "zap" function in the right menu. To save the change across invocations of the graphic editor, execute the "save" function.

6.3 Drawing a Program

You can draw a computer program by simply hooking together icons which you have created on the screen from the second-from-top menu. These icons are the primitive types in the visual language. To connect together instances of these types, simply put the mouse inside one of the 'arguments', the blue squares, of one of the objects on the screen, and press button 3. Then do this to another argument of a different object, and these two arguments will be connected together. You can continue connecting together arguments by connecting already-connected arguments to even more arguments.

You can create an implementation for an abstract type by first drawing the icon for this type, zapping it and saving it. Then, draw its implementation on the screen. To create the gateway between the implementation and the outside world, create "argument" objects on the screen , and modify them to be external arguments by pressing button 4

over them. Then invoke the "abstract" function in the right menu, select the desired abstract icon from the next-from-top menu, and the current program will become the implementation for that icon. You can now use your icon, with its implementation, in drawing your next program.

7 conclusion

The graphic editor for the visual language for parallel processing has been implemented on a Hewlett-Packard Bobcat computer. This graphic editor builds the data structures necessary to fully describe a program in visual format. This program can be written to files in a manner that is transferable between machines, thus allowing compilation or interpretation of the same program by completely different computers, be they SIMD, MIMD, or ordinary, and in addition allowing a program to retain its visual appearance across machines.

The version of the language currently implemented employs all the basic elements of the desired language. It includes aesthetically pleasing graphics, and its relatively easy to use. However, this language does not have the full power of something such as Common Lisp as it stands; however, with the addition of more primitive types and the addition of arrays, along with the capability for the language to compile and interpret its own meta-language and a compiler written for the language in the language, can bring the language up to full-fledged status.

Such a full-fledged language would still be fairly easy to learn and use by novices, and

also be powerful enough for use by experienced computer programmers. By developing such programs on one machine and transferring them to another, a wide variety of programs can be written which are useful in all machines.

The major problem with a visual language is that it is much more difficult to implement, both in terms of a graphic editor and a compiler, than an ordinary text-based language. This obstacle may soon be overcome as the field of computer science advances, allowing an expanded version of this language to become widely used and widely useful.

References

[DiSessa 86a] diSessa, Andrea and Harold Abelson.
Boxer: A Reconstructible Computational Medium.
Communications of the Association for Computing Machinery , September, 1986.

[DiSessa 86b] diSessa, Andrea.
Knowledge in Pieces.
User-Centered System Design: New Perspectives in Human-Computer Interaction , 1986.

[Hanson 87] Hanson, Stephen Jose and David J. Burr.
Knowledge Representation in Connectionist Networks.
Technical Report, Bell Communications Research , Morristown, NJ 07760, February, 1987.

[Kay 84] Kay, Alan.
Computer Software.
Scientific American , September, 1984.

[Liskov 86] Liskov, Barbara and John Guttag.
Abstraction and Specification in Program Development.
MIT Press, 1986.

[MIT 85] MIT Educational Computing Group.
Boxer: An Integrated Personal Computing Environment.
Technical Report, MIT, 1985.

[Sklar 83] Sklar, David.
Piggs: The Polyiconic Interactive Graphical Generation System.
Technical Report, Brown University Computer Science Department, 1983.