

Rx for Syntax: Operator Nets for Formal Visual Programming

W. H. Mitchell
E. A. Ashcroft

Computer Science Department
Arizona State University
Tempe, Arizona 85287

Abstract

A programming environment which provides access to visual as well as traditional textual programming, based on the dataflow paradigm, is presented. This environment, ON, has two faces: a textual programming interface, which allows convenient programming in the dataflow language Lucid, and a graphical portion, which permits visual programming using Operator Nets. Since Lucid and Operator Net programs have the same semantics and are directly interconvertible, we realize a system permitting visual programming but without any sacrifice to the purity of its dual textual language.

1.0 Introduction

Visual programming languages [6][22] have become all the rage, of late, throughout both industry and academia, as, presumably, the hardware to effectively implement such languages has progressed to its current level of sophistication. Frequently cited rationales proffered to explain the visual programming sensation include considerations of the higher information density, culture independence and improved man-machine communication bandwidths possible when employing pictures as program "code".

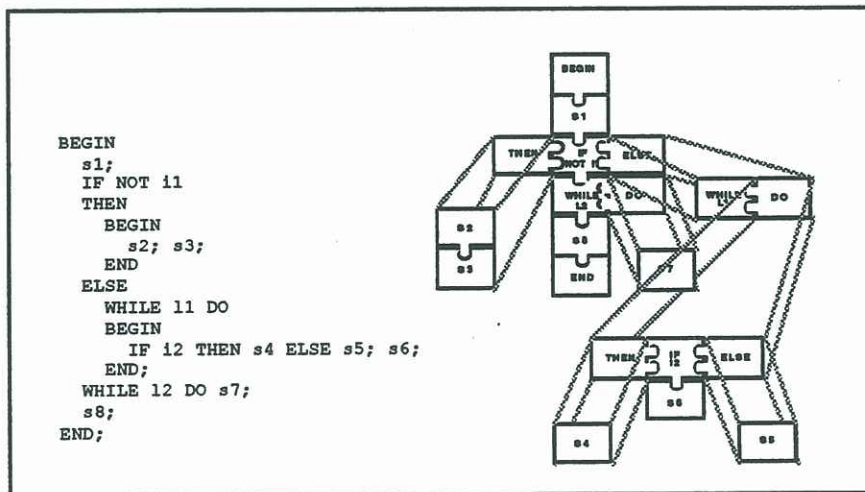


fig. 1

Beyond these advantages directly attributable to the pictorial representations employed in visual programming environments, however, these systems provide other important improvements over text based languages. For example, when first learning a textual language one typically expends much effort learning syntax and tracking down syntactic errors in programs. Most visual languages, by way of contrast, do not even permit the construction of syntactically ill-formed programs. For example, figure 1 below illustrates the source code of a stylized Pascal program and its corresponding *Proc-BLOX* [6] representation. Note that only program units that “fit together” syntactically may be combined to form programs.

In addition to this, some systems perform live type-checking [13] and hence do not even permit the construction of some classes of *semantically invalid* programs.

Further advantages attributable to visual programming systems include the fact that they typically allow related portions of code to be located near each other spatially, whereas this is not normally the case with most present textual programming environments. Also, visual representations admit so-called “2.5-D representations” [6] which make use of encapsulation to reduce screen clutter and provide multiple layers of program construction hierarchy. This is illustrated in the previous figure where it can be seen that certain of the *Proc-BLOX* can be “popped open” to view their lower level component layers.

2.0 Taxonomy of Visual Programming Systems

The term visual programming covers a broad range of the use of visualization in computer programming. Many systems such as PegaSys [20] and VERDI [21] are complete environments which support the use of visual representations over several or a wide variety of areas in program design including documentation, program listings, data-type schema, execution stacks and debuggers. Other visual programming systems deal more exclusively with the direct manipulation of small pictures referred to as *icons* which physically represent portions of a program. These icons are connected to each other by visual connecting links over which typically “flow” either the thread of control or the data elements of the program [20]. Some systems support a more hybrid approach, perhaps, for instance, utilizing “live” message passing agents. Based on these characteristics, we may classify iconic languages as belonging to one of the following subclasses: *control flow-based*, *dataflow-based*, or *object-oriented (OOP) /object based*.

Control flow-based systems are the oldest and probably still the most popular approach to visual programming. One fairly obvious scheme that has been employed has been to exploit the graphical nature of the venerable flow-chart as a direct-manipulation based graphical language. In fact, several successful commercial systems, Matrix Software Technology's *Matrix Layout*, and Mainstay's *V.I.P.* have been based on this simple approach. Glinert and Tanimoto's *Pict* system [10] also uses this control-flow, flowchart-inspired scheme while Glinert's *BLOX* programming methodology [6] appears somewhat less flow-chart like but is still control-flow based.

One common criticism of flowchart-based iconic environments is the “spaghetti-ball” look that problems programmed using these systems tend to acquire, due to large numbers of overlapping interconnections. This problem is further exacerbated when we consider their extension to use within parallel and distributed environments. A classic control-flow based technique that combats this problem is the Nassi-Shneiderman Diagram [19], which coalesces the connections and nodes of a flowchart into embedded graphical units.

A third technique that has been used as a control-flow basis for iconic programming is Augmented Transition Networks (ATNs). Jacob's visual programming system [14] makes use of this approach.

OOP based iconic systems are rapidly gaining in popularity on control-flow based approaches. These systems typically employ a message-passing scheme in which each node in a visual program is an *object* in the OOP sense, which may send messages to other nodes in the program along the "wires" which connect the nodes. Research systems based on this approach include *Hi-Visual* [11][12], Borning's *ThingLab* [5], and Apple Computer's *Fabrik* [13]. Several successful commercial products that also make use of this approach include Serius Corporation's *Serius 89*, National Instrument's *Lab View*, and TGSS's *Prograph*.

The OOP based approach has yielded systems with many interesting features. *Fabrik*, for example, employs a sort of bi-directional, "live" dataflow approach which allows for the elegant solution of certain constraint programming problems, for example, interactive Fahrenheit to centigrade conversion. *ThingLab* also is especially well suited to normally quite complex constraint programming tasks.

The third basis for visual programming systems is the dataflow computing paradigm. Dataflow provides a natural match to iconic programming in that the entire paradigm is based on the visualization of dataflow programs as directed *dataflow graphs* [7]. These graphs consist of nodes which correspond to the operators of the language, and arcs which connect these nodes, which correspond to the data dependencies that exist between operators. Around these graphs flow data *tokens* which are the values computed as the results of operators. Thus, in an iconic dataflow system, icons represent the operators and functions of the program, and graphical lines connecting them represent the intended application of the results computed by one operator to another operator. Pure dataflow iconic systems based on the so called *structure* and *token models* of computation have been constructed by Davis, Keller and colleagues [8][16].

As the reader may have noticed, some iconic languages do not fall cleanly into one classification or the next. For example, several OOP-based systems also combine elements of the dataflow approach. TGSS touts the fact that *Prograph* is dataflow-based, although it also features object oriented characteristics such as classes and inheritance, and *Fabrik*, too, employs an underlying dataflow model of computation.

While control flow-based and OOP-based systems both provide fairly convenient, natural approaches to visual programming, and a number of successful implementations have been built which are based on these paradigms, both suffer from the fact that they have no formal theoretical underpinnings or well defined mathematical semantics. This directly impacts the visual programmer as his programs are subject to side effects and their behavior is difficult to prove correct. Coupled with this is the fact that system scalability and the possibility of concurrent/distributed evaluation is severely limited by the mathematical impurity of these environments.

3.0 Dataflow and Lucid

Dataflow languages form a subgroup of *functional* or *applicative* languages, i.e. those based primarily upon function application. Both functional languages and logic languages are subclasses of the class of *declarative languages*. Although any functional language may be viewed in a dataflow graph manner, the term "dataflow language" is usually reserved for languages specifically designed for execution on dataflow machines. The most well known

of these languages are VAL [2] and ID [2] and Lucid [23]. Lucid in particular offers an exceptionally elegant approach to dataflow computing, and serves as the core language for our research.

Lucid programs, like typical programs in most functional and dataflow languages, are simply mathematical expressions with, possibly, associated sets of defining equations. Program structuring is accomplished through the "where clause" scoping mechanism of Landin [17]. A simple example Lucid program that calculates the distance between two points (x_1 , y_1) and (x_2 , y_2) is:

```
dist where
  dist = sqrt(dxsq + dysq);
  dxsq = (x2 - x1) * (x2 - x1);
  dysq = (y2 - y1) * (y2 - y1);
end
```

We could have written this, by using another **where** clause, more appropriately as:

```
dist where
  dist = sqrt(dxsq + dysq) where
    dxsq = (x2 - x1) * (x2 - x1);
    dysq = (y2 - y1) * (y2 - y1);
  end
end
```

Variables which have no definitions, such as x_1 , y_1 , x_2 , and y_2 , default to program inputs. That is, in the program above, the values of these variables would be requested from the user at run time.

Lucid also incorporates the notion of mathematical functions, as in functional programming. For instance, a slightly modified version of the program above is:

```
dist(x1, y1, x2, y2) where
  dist(a, b, c, d) = sqrt(dxsq + dysq) where
    dxsq = (c - a) * (c - a);
    dysq = (d - b) * (d - b);
  end
end
```

Notice that the variables x_1 , y_1 , x_2 , and y_2 are still input variables, and that variables a , b , c and d are formal parameters of the function **dist**.

Two important features distinguish Lucid from other functional languages: (1) Lucid is *stream-oriented* and (2) Lucid is based on *intensional logic*. Consequently, Lucid programs may be thought of as collections of continuously operating filters through which data elements repeatedly flow. Also, Lucid programs may be given several different readings. Under one *extensional* interpretation of the program below, for example, the program simply adds the values of two scalars a and b . Under the *intensional* interpretation, however, we may view this as a program which performs some operation indicated by '+' on two vectors or matrices (or indeed, any two data objects) to yield a another data object.

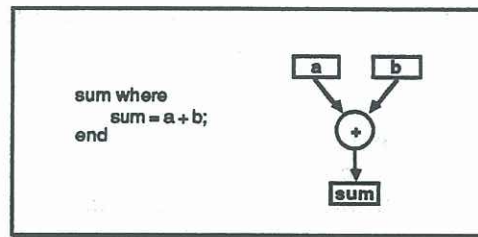


fig. 2

The presence of these two complementary views is one of the strengths of Lucid, since this effectively means that Lucid incorporates features and accrues the benefits of both the *token-model* and the *structure-model*, as per the classification of Davis and Keller [7]. Under either model, the textual program in figure 2 (left) may also be viewed as specifying the *dataflow graph* in figure 2 (right). Under the intensional interpretation we may think of entire vectors flowing as tokens along these arcs, while under an extensional view, individual numbers are data tokens.

4.0 Operator Nets

Lucid has proven to be an elegant and very powerful language. However, due to the linear nature of textual Lucid programs, which is apparently at odds with the multi-dimensional, inherently visual nature of data and data dependencies in these programs, practical programming in Lucid is not always a trivial task. Quite conveniently, however, Lucid happens to be the textual dual of a particular dataflow graph programming language: *Operator Nets* [4]. An Operator Net is a type of uninterpreted dataflow-graph-based graphical language which may be given different interpretations depending on the sequence algebra on which it is based. (The particular algebra we will be concerned with in this section is identical to that of the language *pLucid*.) As defined by Ashcroft and Jagannathan in [4] [15], Operator Nets consist of a main directed graph together with subsidiary directed graphs, whose nodes may be classified as illustrated in the table below.

| Node Type | Node Description |
|------------------------|--|
| <i>Operators</i> | functions of the sequence algebra of the language |
| <i>Functions</i> | user defined functions with defining operator nets |
| <i>Subcomputations</i> | nodes that enclose subsidiary operator nets |
| <i>Forks</i> | nodes which duplicate data |
| <i>Inputs</i> | input nodes of an operator net |
| <i>Outputs</i> | output nodes of an operator net |

fig. 4

The operator net visual programming environment we have developed takes certain liberties with regards to this definition. For instance, to ease programmer burden we allow *fan-out* from operators, functions, and input nodes instead of forcing the programmer to explicitly use *fork* nodes. Also, in our environment we essentially coalesce function nodes

and subcomputation nodes, treating an icon for a function to be an object which may be “popped open” (via suitable visual manipulation: a mouse “double-click” on the function icon), revealing its defining subgraph. In addition to these changes, we employ icons slightly different to the ones first described by Ashcroft and Jagannathan for the sake of implementation grace and convenience.

As an example of what our slightly modified operator nets look like, we show a very simple program that computes and returns a value that is six times the input value the user supplies plus six, and a sample operator net program that computes the same result as this Lucid program (figure 5).

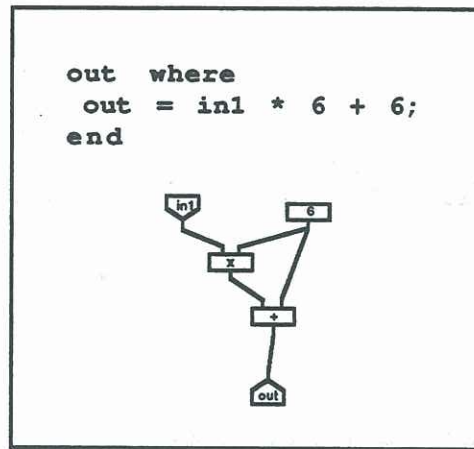


fig. 5

5.0 ON: The Operator Net Programming Environment

We have created an implementation of operator nets which we call *ON*, to allow experimentation with dataflow-style iconic programming and more traditional, text-based dataflow programming, side-by-side. This environment is centered around an interactive, direct-manipulation user interface and runs under UNIX and the X Window system.

To facilitate further discussion of *ON*, we present the operator net program from figure 5, but now in the *ON* environment, in figure 6.

ON provides access to all commands affecting the environment through menu selection from the menu bar at the top of the main window. The balance of the main window contains a “programming canvas” on which users may connect together operator icons to form programs. Operator icons (e.g. ‘x’, ‘+’ in figure 6) are “dropped” onto the canvas in response to appropriate operator menu selections. They are connected together via mouse manipulation: “mouse-down” on one icon’s input/output prong, “drag” to another icon’s output/input terminal, then “mouse-up”. Program icons may be moved freely by “dragging” them with the mouse, whether or not they are connected.

The *ON* environment also contains all of the amenities that users of visual environments such as this one have a right to expect. These include the ability to “cut” and “paste” operators, control over the canvas real estate through pans and zooms, and mechanisms for printing operator net windows.

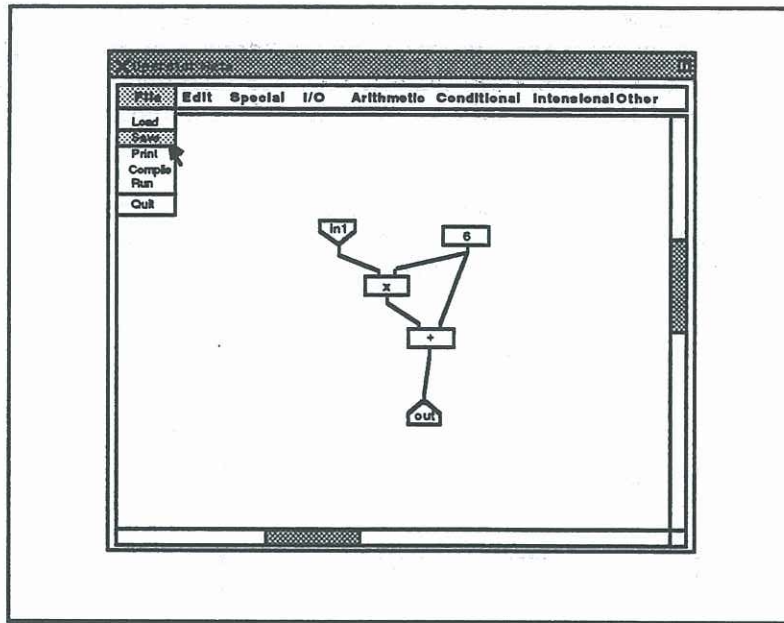


fig. 6

One of the more interesting and unique features of ON is its treatment of user defined functions. Users may create their own functions by selecting “U.D.F” from the menu, then programming on a separate canvas specific to that function.

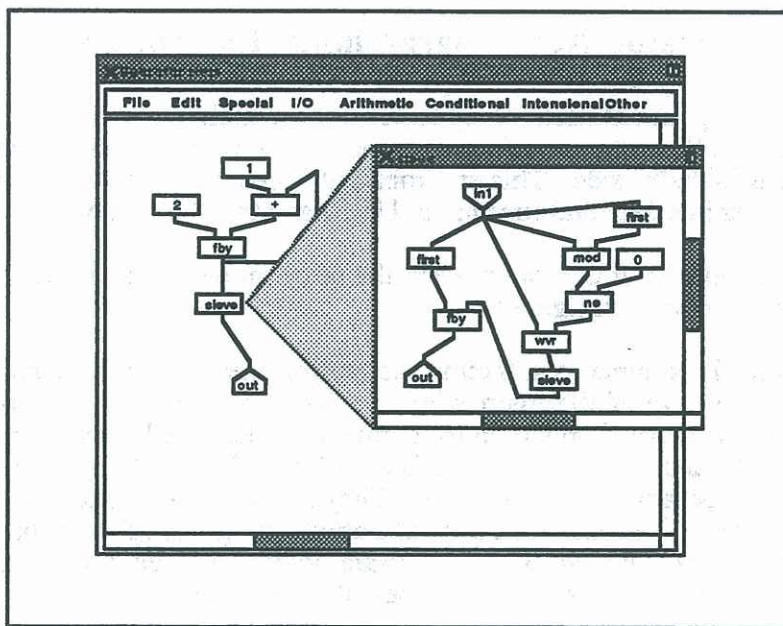


fig. 7

Figure 7 above shows an example of this: a program that computes prime numbers using the Sieve of Eratosthenes technique. Notice that the **sieve** user-defined function has been “popped-open” to reveal its defining (recursive) sub-graph.

Another particularly useful feature of this system is its ability to automatically convert from Lucid text to operator net form, and vice-versa. This capability, coupled with the inclusion of a text-editor in ON, serves to allow limited, mixed textual and visual programming in a single convenient environment. As a result, advantages of both textual and visual modes of programming are realized at once, and the user is not shoehorned into one particular syntactic style.

An example of the auto-generation of Lucid text from operator nets, a follow-on to figure 7, after the user has presumably selected "compile text" from the menu, is illustrated below.

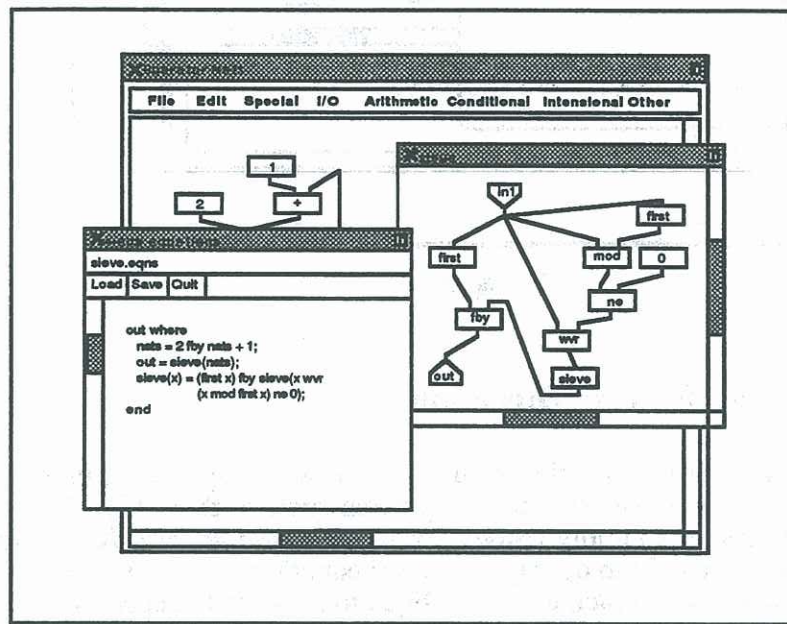


fig. 8

Key to the whole environment is the Lucid interpreter which evaluates operator net and Lucid programs. Interpretation may be invoked via a menu option, as usual. Since the interpreter only accepts textual input, if the program is in operator net form it is first transparently converted to Lucid text. After this, an interpreter window is opened, any input data needed for the program is requested, and results of the program are displayed. Since, as discussed in section 3, Lucid is stream-based, outputs are continuously generated until the user signals he has seen enough. The figure below completes our sequence of illustrations by showing a "run" of the `sieve` program.

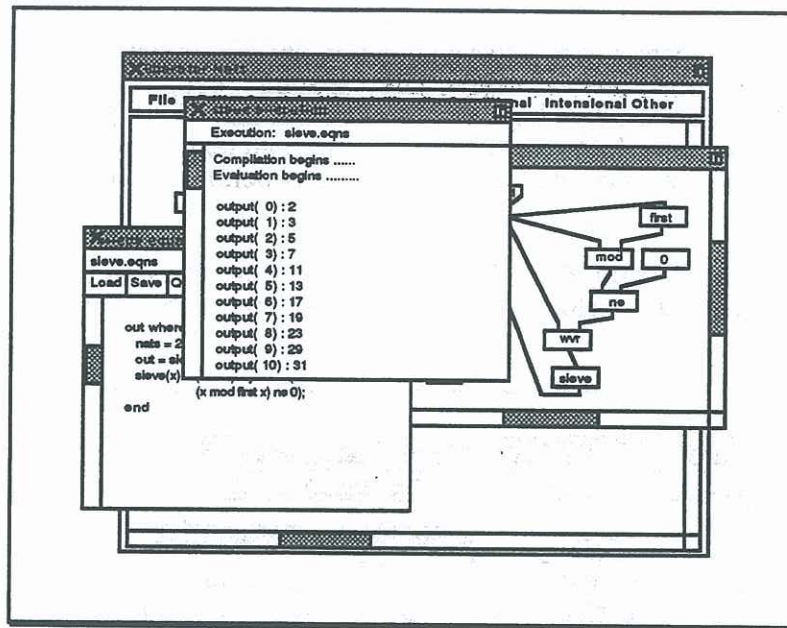


fig. 9

6.0 Summary and Future Directions

The operator net editing canvas portion of the system was constructed using a generic platform for prototyping graph-based object systems such as this [18]. Currently, the Lucid equation editor module simply consists of a call to the generic X Window editor *XEdit*. The translators to and from operator nets and equations are the subject of the most intense ongoing research, and, in fact, the Lucid Equation-to-Net Translator module has yet to be implemented. While simple translations in either direction have proven quite easy to perform, complex issues such as automatic routing, treatment of recursion and handling of higher order functions have provided set of research problems.

A dual textual/visual environment for the construction of dataflow programs has been designed and and partially implemented. It provides features and advantages to be found in other commercial and academic visual programming environments such as a direct-manipulation, "point-and-click" style user interface and menus of operators which may be extended with user defined functions as well as several unique features including automatic text-to-graph and graph-to-text translations, and a tagged, demand-driven evaluation scheme.

To make the system fully functional, additional work remains to be done. This includes the full development of the User Defined Function feature, the addition of routing "vertices" to the current point-to-point connection scheme, and the development of the Equation-to-Net Translator module.

Additional future work on this system will possibly involve adding on weak, connect-time type-checking extensions, the design and construction of an interpreter that evaluates operators nets directly, and the extension of the evaluation mechanism for use in distributed and concurrent systems.

References

1. W. B. Ackerman, "Data Flow Languages," *IEEE Computer*, vol. 26, no. 2, pp. 15-25, Feb. 1982.
2. Arvind and R. S. Nikhil, "Executing a Program of the MIT Tagged-Token Dataflow Architecture," in *IEEE Trans. on Computers*, vol. 38, no. 3, pp. 300-318, March 1990.
3. E. A. Ashcroft, "Dataflow and Education: Data-driven and Demand-driven Distributed Computation," Tech. Rep. CSL-151, SRI International, Menlo Park, CA., March 1986.
4. E. A. Ashcroft and R. Jagannathan, "Operator Nets," Tech. Rep. CSL-150, SRI International, Menlo Park, CA., March 1986.
5. A. Borning "The Programming Language Aspects of ThingLab, A Constraint Oriented Simulation Laboratory," in *ACM TOPLAS*, vol. 3, no. 4, pp. 353-387, October 1981.
6. S. K. Chang, Ed., *Principles of Visual Programming Systems*. Englewood Cliffs, N.J.: Prentice Hall, 1990.
7. A. L. Davis and R. M. Keller, "Data Flow Program Graphs," *IEEE Computer*, vol. 26, no. 2, pp. 26-41, Feb. 1982.
8. A. L. Davis and S. A. Lowder, "A Sample Management Application Program in a Graphical Data-Driven Programming Language," in *Digest of Papers of Compcon Spring 81*, pp. 162-167, Feb. 1981.
9. M. Foley, W. Mitchell, A. Bose, A. Faustini, and J. Singh, "An object-oriented man machine interface for power systems," submitted to *PICA 91 Proc.*, May 1991.
10. E. P. Glinert and S. L. Tanimoto, "PICT: An Interactive, Graphical Programming Environment," *IEEE Computer*, vol. 17, no. 11, pp. 7-25, November 1984.
11. M. Hirakawa, M. Tanaka, and T. Ichikawa, "An Iconic Programming System: Hi-Visual," *IEEE Trans. Software Eng.*, pp. 1,178-1, 184, October 1990.
12. T. Ichikawa, and M. Hirakawa, "Iconic Programming: Where to Go?," *IEEE Software*, vol. 25, no. 9, pp. 63-68, November 1990.
13. D. Ingalls, S. Wallace, Y. Y. Chow, F. Ludolph, and K. Doyle, "Fabrik, A Visual Programming Environment," in *OOPSLA 88 Conference Proc.*, Sept. 1988.
14. R. J. K. Jacob, "A State Transition Diagram Language for Visual Programming," *IEEE Computer*, vol. 18, no. 8, pp. 51-59, August 1985.
15. R. Jagannathan, "A Descriptive and Prescriptive Model for Dataflow Semantics," Tech. Rep. SRI-CSL-88-5, SRI International, Menlo Park, CA., May 1985.
16. R. M. Keller and W. C. J. Yen, "A Graphical Approach to Software Development Using Function Graphs," in *Compcon Spring 81*, pp. 156-161, Feb. 1981.
17. P. J. Landin, "The Next 700 Programming Languages," *Comm. ACM*, vol. 9, no. 3, pp. 157-166, March 1966.
18. W. H. Mitchell et. al., "A Platform for the Rapid Development of Object-Oriented Net-Based Systems," submitted to *OOPSLA '91*, Oct. 1991.
19. I. Nassi and B. Shneiderman, "Flowchart Techniques for Structured Programming," *ACM SIGPLAN Notices*, vol. 8, no. 8, pp. 12-26, August 1973.
20. G. Raeder, "A Survey of Current Graphical Programming Techniques," *IEEE Computer*, vol. 18, no. 8, pp. 51-59, August 1985.
21. V. Y. Shen, C. Richter, M. L. Graf, and J. A. Brumfield, "VERDI: A Visual Environment for Designing Distributed Systems," *Journal of Par. and Dist. Comp.*, vol. 9, pp. 128-137, September 1990.
22. N. C. Shu, *Visual Programming*. New York: Van Nostrand-Reinhold, 1988.
23. W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language*. London, England: Academic Press, 1985.