Representing Visual Programs with Object-Graphs

Eric J. Golin
Steven P. Reiss

Brown University
Department of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-89-05

February, 1989

Representing Visual Programs with Object-Graphs *

Extended Abstract

Eric J. Golin†
Steven P. Reiss
Brown University
Department of Computer Science
Providence, Rhode Island 02912

February, 1989

Abstract

We consider the development of a representation for programs in a visual programming environment. Abstract Syntax Trees are generally used to represent textual programs, but tree based structures are inadequate as a basis for representing visual programs. We generalize abstract syntax trees to abstract syntax graphs and introduce Object-Graphs as a definition. We show how object-graphs can be used to represent visual programs.

1 Introduction

Traditional programming languages are *textual*, and programs are text strings. *Visual languages* [1] are two dimensional, and use pictures to form programs. Visual languages promise to enhance programmer productivity by providing a more comprehensive medium than traditional textual languages. Pictures allow the presentation of more information than text in an accessible manner. Because pictures are already two dimensional, manipulating pictures within a visual environment is more natural than manipulating text with a syntax directed editor.

A basic principle of visual programming is to have the pictures be the program. This implies that the program representation must correspond to the pictures. When the user manipulates the pictures, he is actually acting on the underlying program representation. The program representation must serve as a basis for defining the execution of the program.

Whereas text is essentially one dimensional, pictures are inherently multidimensional. For example, Figure 1 shows a sample program in a visual language based on finite state automata. The picture of the FSA expresses the complex relationships between the components in a natural, direct manner. A two dimensional display can show many connections at once. Additional dimensions can be achieved by utilizing refinements of the displayed entities or presenting several pictures.

*This research was supported in part by the Office of Naval Research and the Defense Dept.'s Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA Order No. 4786, by NSF grant SER80-04974, by a contract with IBM, by a grant from AT&T Foundation, and by a grant from Digital Equipment Corp. Partial equipment support was provided by Apollo Computer, Inc.

†Support for this author was provided by an I.B.M. Graduate Fellowship

Our goal is to define a program representation that will serve as a basis for a language-independent environment for visual programming. [2] A visual programming environment would allow a user to define a visual programming language and then create and execute visual programs. The environment is based on an abstract representation for programs, as shown in Figure 2. The abstract representation serves as a basis for both a syntactic and a semantic mapping. This approach is similar to that used in environments for textual programming languages, [3] [4] [5] where *abstract syntax trees* are used as a program representation.

The multidimensional nature of visual languages makes abstract syntax trees a poor representation for programs. The use of a tree structure distorts the syntactic and semantic mappings. Graph structures are more general than trees, so *abstract syntax graphs* provide a better mechanism for representing visual programs. This paper discusses why abstract syntax graphs are preferable to trees and introduces *object-graphs* as a formal definition of abstract syntax graphs. An example is given of how object-graphs can be used to represent visual programs.

2 Program Representations

A program representation must take into account all aspects of the language to be represented. These features can be broken down into four components

- The concrete syntax.
- The abstract syntax.
- The static semantics.
- The dynamic semantics.

The *concrete syntax* describes the program as it appears to the user. The *lexical* structure is the lowest level of the concrete syntax, and consists of the basic display items, such as characters or lines. These lowest level items are grouped into lexical elements. The concrete syntax says what strings of lexical elements form programs in the language.

For visual languages, the concrete syntax is one or more pictures. A picture consists of distinct graphical elements, just as a textual program consists of a string of lexical elements. These graphical elements may be combined to form larger components of the picture. In this manner, the picture can be decomposed into a hierarchy of components. In addition to the hierarchical decomposition into components, a picture will contain other relationships between entities. For example, the nodes of a data flow graph have connections between them which reflect input-output relationships. The representation must be capable of expressing both kinds of constructions.

The *abstract syntax* embodies the underlying structure of the language. The representation of a program is based on the abstract syntax of the language. The abstract syntax specifies what constitute syntactically valid program structures, for which the concrete syntax determines the displayed form.

The *static semantics* of the language deals with what a program says. The static semantics express the context sensitive properties of the language. For example, the notion that the operands of an expression must have types compatible with the operator is an instance of static semantics. In a visual language, the static semantics can be viewed as a restriction on the structures of pictures that may be created. For a visual expression language an analog to the previous example would be the restriction that an operator node may only be drawn connected to compatible operand nodes.

Finally, the representation must also serve as a basis for the *dynamic semantics* of the language. Dynamic semantics is concerned with what the program does when it is executed. The dynamic semantics could be

provided for in several ways. For example, the representation could be used to compile a program into an executable form. Or the representation could be used to interpret the program. Neither of these can be done directly with the simple textual representation of the program. For a visual programming system to be useful, the execution of pictures must be defined. If the user introduces new languages corresponding to his conceptual models, he must be able to specify how the pictures in those languages are executed.

A good representation for programs should do the following:

- Embody the underlying structure (abstract syntax) of the language.
- Incorporate the static semantics of the language.
- Provide for mapping between the concrete and abstract syntax of the language.
- Provide a basis for the dynamic semantics of the language.

3 Abstract Syntax Trees

Textual languages are often defined using some form of context free grammar. The grammar specifies both the concrete syntax and the underlying structure of the language. The underlying structure of a program is expressed by its parse tree.

Programming environments for textual languages typically represent programs using *abstract syntax trees* [3] [4] [5] [6] An abstract syntax tree (AST) is a generalization of the parse tree and reflects the context-free grammar underlying the language. The leaf nodes of the tree correspond to terminal symbols in the grammar. Internal tree nodes are related to the non-terminals of the grammar, but the parse tree is transformed into a canonical form to create the AST. For example, the rules

$$St\_list \Rightarrow St; St\_list$$

$$St\_list \Rightarrow St$$

when applied repeatedly would generate a tree with many *St_list* nodes. A canonical form of this would have a single *St_list* node with many *St* nodes as children, as in Figure 3. Using an abstract syntax tree rather than the parse tree frees the environment implementor from the minutiae of the concrete syntax, while still retaining all the information of the program.

Tree structures provide a natural base for defining syntactic mappings. Mapping from the concrete syntax into an AST is a simple extension of parsing to construct the AST instead of the parse tree. Unparsing, or mapping from the AST back to the concrete syntax, is also simple to perform. [1] The text for a node in the AST is constructed from the text for its children nodes. The naturalness of these mappings is due to the linear nature of the tree representation.

Semantics can be assigned to AST's in a number of ways. One common approach is to use attribute grammars. [5] [7] This approach associates a set of attributes with each node in the tree. Each production in the grammar has a set of semantic equations to define attributes. The attributes of a node will either be defined by the node above it (for inherited attributes) or by the attributes of the its children nodes (for synthesized attributes). Information can flow both up and down the syntax tree.

Attribute grammars are most useful for expressing the static semantics of programs. A requirement that variables be declared is an example of language constraint based on the context in which a construct appears.

[1] Although producing the complete text of the program is straightforward, there are problems such as elision that complicate the determination of the text to be displayed in an environment.

Attributes embody this context. Attribute grammars are a good tool for specifying such constraints because they are modular (i.e. the attributes of a node are determined locally) and based on the syntactic structure (the AST).

Another approach to specifying semantics is the use of action routines. [8] The implementor writes action routines, which are associated with rules in the syntax description. Each node in the AST is associated with the rule for the corresponding production. The action routines contain code to determine what command is being executed and take appropriate action. Action routines provide a flexible mechanism for describing dynamic semantics. A refinement of the action routines paradigm is *action equations*, which combine action routines with attribute grammars. [9]

Denotational semantics is another method of specifying semantics of programming environments. [10] The denotational definition of the semantics assigns a program a meaning in an abstract domain. A metalanguage is used to express the semantics for the source language. Each construct in the abstract syntax is defined in terms of its components. A program can then be executed using an interpreter for the metalanguage.

The common thread of these methods of specifying language semantics is that they are based on the tree structure of the abstract syntax. They utilize the hierarchy to construct the semantics of a node in terms of its children. Using the hierarchy provides a natural definition of semantics for textual languages because of their tree structure.

Abstract syntax trees perform well for those aspects which conform to the tree structure, but do not provide for non-hierarchical features. This restriction has several important shortcomings:

- Visual languages are two-dimensional and generally do not have a tree structure. For example, consider the finite state automata language shown in Figure 1. Programs in this language can contain complex relationships between states that cannot be expressed by trees. To construct an AST for an FSA, the structure must be coerced into a tree such as that shown in Figure 4. This AST does not reflect the actual underlying structure of the program.

- References between nodes in the tree must be local (i.e. between adjacent nodes). Nodes which are closely related semantically, such as the declaration and use of a variable, may be arbitrarily far apart in the tree.

- Not all semantic relationships correspond to the tree structure. For example, the AST for a Pascal program does not represent any data-flow information about the program. [11]

It is clear that a representation for graphical programs must be based on a structure more general than trees. Several systems have used graphs as a basis for representing programs. [12] [13] [11] Graphs allow the natural connections between components to be expressed. For example, a graphical data flow language can be mapped directly into an underlying graph-based structure. A tree representation would either lose or hide this structural information. Since references in graphs are not restricted to syntactically adjacent nodes, no distinction is made between local and non-local relationships. Graph structures provide a more uniform access method for information. In the next section we introduce the object-graph as a general graph-based structure to address these problems.

4 Object Graphs

Object-graphs are an object-oriented representation based on graph structures. They serve as the abstract syntax for a visual language. Object-graphs provide a natural mechanism for representing visual programs. There are three kinds of object-graphs. Primitive object-graphs represent atomic elements from the underlying domains such as integers or strings. A list object-graph combines a list of object-graphs of the same

type. Finally, complex object-graphs are used for structured objects, and consists of a collection of named fields. The fields correspond to the instance variables of an object.

We define object-graphs over $\mathcal{D}$, a set of primitive domains, $\mathcal{N}$, a set of identifiers and T, a set of object-graph types.

Definition 1 *A* **primitive object-graph** *is a tuple* $POG =< t, v >$ *where* $t \in T$ *is the type of the object-graph and* $v \in \mathcal{D}_t$ *is a value in the corresponding primitive domain.*

Definition 2 *A* **list object-graph** *is a tuple* $LOG =< t, L >$, *where* $t \in T$ *is the type of the object-graph and* L *is either* NULL *or a list of object-graphs.*

Definition 3 *A* **complex object-graph** *is a tuple* $COG =< t, F >$ *where* $t \in T$ *is the type of the object-graph and* F *is given by* $F = \{< f_i, v_i >\}$ *where* $f_i \in \mathcal{N}$ *is a field name and* v_i *is an object-graph.*

Definition 4 *An* **object-graph** *is either*

1. *a primitive object-graph*
2. *a list object-graph*
3. *a complex object-graph*

Object Graphs are similar to other notions of objects such as Smalltalk, [14] or Lisp Flavors. [15] Object-graphs are instances, that is, actual objects representing specific programs. Object-graph types correspond to the notion of class. An object-graph system defines a complete set of object-graph types.

We will generally just show a primitive object-graph as its value, unless the type is not apparent, when we will use the notation *value*$\diamond$*type*. For a list object-graph, we will use the notation $(og_1, og_2, \ldots)\diamond type$, as in (1, 2, 3)$\diamond$IntList. A complex object-graph is displayed $\langle field_1{:}og_1,\ field_2{:}og_2,\ \ldots\rangle\diamond type$. The type of a list or complex object graph may also be omitted if obvious.

An object-graph system defines a related set of object-graph types. The type of a simple object-graph is defined by specifying the underlying domain. The type of a list object-graph is defined by the type of the elements of the list. The type of a complex object-graph is defined by a set of tuples defining the fields of the complex object-graph.

Definition 5 *An* **object-graph definition** *over* T *and* $\mathcal{N}$ *is given by* $D_{OG} = \{< f_i, t_i >\}$ *where* $f_i \in \mathcal{N}$ *is the field name and* $t_i \in T$ *is the type of the field.*

Field names are unique for an object-graph definition (so $f_i \neq f_{i'}$ for $i \neq i'$), but distinct definitions may have fields with the same name.

Definition 6 *An* **object-graph system** *over the set of domains* $\mathcal{D}$ *and identifiers* $\mathcal{N}$ *is given by* $OGS = (T, S, D, C)$ *where*

- $T \subseteq \mathcal{N}$ *is the set of types for the OGS.*
- $S \subset T \times T$ *is the* supertype relation *for the OGS. S defines a partial order over T. If* $< t_1, t_2 > \in S$, *then we write* $t_1 \sqsubseteq t_2$ *to mean that* t_2 *is a supertype of* t_1.
- D *is a set of object-graph definitons over* T *and* $\mathcal{N}$.
- $C : T \rightarrow D \cup T \cup \mathcal{D}$ *is the* type definition function. *If* $C : t_1 \rightarrow \mathcal{D}$, *then* t_1 *is a primitive object-graph type. If* $C : t_1 \rightarrow T$, *then* t_1 *is a list object-graph type. Otherwise* $(C : t_1 \rightarrow D)$, t_1 *is a compound object-graph type.*

An object-graph system is similar to a complete object system. The types correspond to the classes, and C and D specify the class definitions. The supertype relation is similar to an inheritance relation, but is used for type consistency as described below, rather than for definition. The notation used to define an object graph system is shown in Figure 5. The supertype relation is defined by specifying the supertypes (if any) for each type. A primitive object-graph $POG =< t, v >$ is *consistent* with an object-graph system OGS if $C : t \rightarrow d \in \mathcal{D}$ and $v \in d$. A list object-graph $LOG =< t, l >$ is consistent with an object-graph system OGS if $C : t \rightarrow t' \in T$ and $\forall v$ in $l, TYPE(v) \sqsubseteq t'$ and v is consistent with OGS. A compound object-graph, $OG =< t, F >$ is consistent with an object-graph system OGS if $C : t \rightarrow D_{OG} \in D$ and $\{f_i \mid< f_i, v_i >\in F\} = \{f_i' \mid< f_i', t_i >\in D_{OG}\}$, and $\forall f_i \mid< f_i, v_i >\in F$, we have $TYPE(v_i) \sqsubseteq t_i$ and v_i consistent with OGS. We shall assume that all object-graphs used to represent programs are consistent with their defining object-graph system.

We define a special domain $\mathcal{NULL} = \{\perp\}$, and a corresponding primitive object-graph type NULL. An instance of type NULL is a primitive object-graph, **Null** $= \perp \diamond$NULL. The type NULL is a subtype of all other object-graph types, so NULL $\sqsubseteq t \forall t \in T$ and if $t \sqsubseteq$ NULL, then $t =$ NULL. In our examples of object graphs, we will assume primitive object graph types are defined for typical domains such as integers, booleans and strings.

5 Using Object-Graphs

This section illustrates how object-graphs are used to represent programs. As discussed in section 2, a representation must embody the abstract syntax, include the static semantics, provide for the concrete syntax and serve as a basis for the dynamic semantics. Object-graphs must account for four aspects of visual language structure:

- The hierarchical structure of the language.
- Within a level of the hierarchy, the graph structure of the language.
- References between levels of the hierarchy.
- The (static) semantic restrictions on the structure of the object-graph.

Tree based structures create hierarchies by encapsulating program fragments into subtrees. Object-graphs provide two mechanisms for implementing hierarchy: complex object-graphs and list object-graphs. Figure 6 shows an object-graph system for the FSA language with an hierarchical structure similar to that shown in figure 4. An FSA is comprised of a name and lists of arcs and states. The list object-graph types ArcList and StateList allow similar objects to be grouped as a single component. A state consists of a name and an accept flag. By viewing the fields of a complex object-graph as *tree edges*, the hierarchical structure of the language is expressed.

The hierarchical structure of a visual language divides a program into discrete levels. Within each level, the structure will be a graph. The FSA language is divided into three levels: the level of the entire FSA, the set of states, and for each state, its components. Within the set of states, there is a graph structure formed by the transitions between states. These transitions are implemented by the Arc object-graph type. Each Arc object-graph represents a connection between two states. In this implementation, all the connections for the graph (i.e. the FSA) are collected into one list of arcs.

Another approach to representing the connections is to attach the arcs directly to the states. Figure 7 shows an alternate object-graph representation for the FSA language where each state contains a list of arcs leaving that state. The arcs only need to contain the matching input character and the next state.

This example also shows a different type of grouping. The FSA object-graph contains a pointer to only the starting state. The rest of the states comprising the FSA are found by following the arcs from each state to the next states.

In the FSA language, the arcs carry information in addition to connection between two objects (namely, the input character for the arc). If the connection is all that is to be represented, this can be done directly with a field of a complex object-graph. For example, Figure 8 defines object-graph types for a simple statement language. Each statement object-graph points to the statement(s) which follow it.

An object-graph may contain references other than those corresponding to either the hierarchy or the graph structure within a level of the hierarchy. For example, the FSA object-graphs defined in Figures 6 and 7 contain a reference to the *current* state. This reference does not really reflect part of the hierarchy or the graph structure of the FSA.

The static semantics of a visual language manifest themselves in restrictions on the structure of the object-graphs. These restrictions are implemented by the type system of the object graphs. The *consistency* property of object-graphs insures that, for example, an Arc of an Fsa points to a State object-graph. The supertype relation allows several object-graph types to be grouped together to provide a union type. For example, the statement language in Figure 8 groups assignment, conditional and return statements together. This allows any type of statement to follow any other type of statement. An InitStmt object-graph can only begin a block however, since InitStmt is not a subtype of Statement. The type system of the object-graph system serves a similar role as the context-free grammar in defining the permitted structures.

5.1 Relating Object-Graphs to Pictures

The concrete syntax for visual languages consists of pictures. For object-graphs to form an effective representation for graphical programs, we must provide mechanisms for mapping between the pictures and the corresponding object-graphs. Mapping from an object-graph to its visualization is the problem of *generating* a picture. Conversely, mapping from a picture to the corresponding object-graph involves *recognizing* the picture. There are several approaches to both these problems.

One approach is to directly define mappings from the object-graph types to a set of graphical objects. The *Gelo* system [16] is a mechanism for defining such mappings. Gelo allows the definition of graphical objects which will form the pictures. Gelo supports three styles of drawing pictures:

- *data objects* consist of a shape containing text.
- *tiled objects* are a rectangular region subdivided into non-overlapping rectangular tiles, each of which contains another graphical object.
- *layout objects* consist of nodes connected by arcs. The nodes are laid out inside a rectangular box according to various heuristics.

The APPLE editor defines mappings from each object-graph type into Gelo objects. Several mappings may be provided for an object-graph type, with conditions attached to select the appropriate mapping. If a complex object-graph is mapped into a tiled Gelo object, the fields are used for the contents of the tiles. When mapping into a layout object, the user specifies which fields represent the contents of the layout, and whether object-graphs are arcs or nodes. This specification provides an interpretation of the structure of the object-graph. Once the mappings are defined for an object-graph system, an actual object-graph is mapped into a Gelo object. Figure 1 shows an FSA program displayed by Gelo.

Mapping from the pictures to the underlying object-graphs is accomplished using a structure editor. The editor maintains a correlation between the Gelo objects and the object-graphs. An operation on the picture

such as inserting an object is mapped into an operation on the underlying object-graphs. The language designer must define the meaning of the editing operations with respect to the object graphs. Changes to the structure of the object-graph are then reflected in the display.

This approach to graphical syntax permits an easy definition of the pictures, but has several drawbacks. The first is that most of the details of the picture (e.g. size, layout position, etc.) are determined by system, and the user has limited control over them. A more serious problem is that the Gelo approach does not provide a basis for recognizing (i.e. parsing) pictures. The structure editor is too restrictive to allow a natural style of editing visual programs.

5.2 Semantics with Object-Graphs

Specifying the semantics of a visual language requires defining the execution of the pictures. Object-graphs provide a basis for such definitions. There are several approaches to defining execution.

The operational approach to defining execution is to define an evaluation mechanism for object-graphs. The simplest approach is similar to the action routine mechanism. Each object-graph type has an associated evaluation routine. The routine specifies evaluation for an object-graph in terms of its components (i.e. the value, elements or fields of a primitive, list or complex object-graph, respectively). These components may in turn be evaluated or may be treated as values.

For example, Figure 9 gives a definition for the evaluation of the FSA language defined in Figure 7. An Fsa object-graph is evaluated once for each input character, starting with the special character `START_SYMBOL`. Each evaluation returns a flag indicating whether the Fsa is in an accepting state. The Fsa executes by evaluating the current state object-graph with the input character. The State object-graph traverses its list of arcs, evaluating each with the input character. The Arc evaluates by comparing the input character with its matching character. If the Arc matches, the State will return the next State for that Arc.

The execution of an object-graph can also be defined directly by the object-graph system. A set of object-graph types with evaluation defined as described above can form an abstract machine. An evaluation mechanism can then be developed based on this machine. This mechanism can take the form of message passing as in Smalltalk,[14] or of function evaluation as described above.

Another approach to execution is to compile an object-graph into a program in another language. The language designer must specify how each object graph is to be translated, generally in terms of its components. For instance, the designer of the statement language in Figure 8 could define a translation from the object-graphs into a textual language such as Pascal. This approach is only effective if there is an existing language with a model of computation similar to that of the visual language.

Finally, a general method of specifying the semantics is to define a universal computational model and then provide mappings from the object-graph system into the semantic model. Just as with the syntactic mappings, an object-graph has a corresponding set of semantic objects within the semantic model. The execution of the semantic objects is defined by the universal model.

6 Conclusion

We have shown how object-graphs provide a better basis for representing visual languages than abstract syntax trees. Object-graphs can also be used as an internal representation for textual languages. The flexibility of object-graphs allow them to naturally represent all features, including non-local references, of the program. We have used the object-graph approach to implement the Pascal language.

The Garden system is an object-oriented environment for visual programming that embodies the object-graph approach. The underlying object system HUMUS supports the definition and creation of object-

graphs. The THYME and VIOLET editors are built on the Gelo/APPLE/PEAR packages and provide a mechanism for defining the visual syntax of programs, displaying programs and editing them. The object evaluation mechanism allows an operational semantics to be defined for the language. The Garden environment has been used to prototype a number of visual languages including languages based on data-flow diagrams, flowcharts and the Paisley executable specification language.

Future work should build on the object-graph approach by developing better mechanisms for defining the syntax and semantics from the object-graphs. We are currently investigating a grammar-based approach to syntax, [17] [18] which will resolve the problems with the Gelo approach. A *picture grammar* is used to define the pictures. The grammar has shapes as terminal symbols and production rules corresponding to operators for combining shapes. The grammar symbols carry attributes to specify features such as size and position. The productions specify constraints on these attributes. An object-graph is mapped by a series of productions into a collection of terminal shapes and a set of constraints. The constraints are solved to generate the actual picture.

The grammar defining the pictures can be used to parse a set of graphical objects. Recovering the structure of the picture allows the corresponding object-graph to be constructed. A graphical parsing mechanism can be used in conjunction with a general purpose graphics editor to provide a natural interface for inputing visual programs.

Much research is also left to be done on developing a consistent semantics for object-graphs. First an appropriate compuational model must be defined, and then a mechanism for specifying mappings from the object-graph into the model must be developed. Object-graphs form a sold base for the development of visual programming environments.

References

[1] Georg Raeder. A survey of current graphical programming techniques. *IEEE Computer*, 18(8):11–25, August 1985.

[2] Steven P. Reiss. Working in the garden environment for conceptual programming. *IEEE Software*, 4(6):16–27, November 1987.

[3] Steven P. Reiss. *PECAN: Program Development Systems that Support Multiple Views.* Technical Report CS-83-29, Brown University, 1983.

[4] Raul Medina-Mora. *Syntax-Directed Editing: Towards Integrated Programming Environments.* PhD thesis, Carnegie-Mellon University, 1982.

[5] Thomas W. Reps. *Generating Language-Based Environments.* The MIT Press, 1984.

[6] Michael A. Caplinger. *A Single Intermediate Language for Programming Environments.* PhD thesis, Rice University, 1985.

[7] C.N. Fischer, Gregory F. Johnson, Jon Mauney, Anil Pal, and Daniel Stock. The Poe language-based editor project. In *Symposium on Practical Software Development Environments*, pages 21–29, ACM, April 1984.

[8] Vincenzo Ambriola, Gail Kaiser, and Robert J. Ellison. *An Action Routine Model for Aloe.* Technical Report CS-84-156, Carnegie-Mellon University, August 1984.

[9] Gail Kaiser. *Semantics for Structure Editing Environments.* PhD thesis, Carnegie-Mellon University, 1985.

[10] R. Barbuti et al. Programming environment generation based on denotational semantics. In D. Ferrari, M. Bolognani, and J. Goguen, editors, *Theory and Practice of Software Technology*, pages 51–83, North-Holland, 1983.

[11] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Symposium on Practical Software Development Environments*, pages 177–184, ACM, April 1984.

[12] Manfred Nagl. An incremental programming support environment. In *Europhysics Conference on Software Engineering, Methods and Tools in Computational Physics*, North-Holland, Brussels, August 1984.

[13] Hausi A. Müller. *Rigi - A Model for Software System Construction, Integration and Evolution based on Module Interface Specifications.* Technical Report COMP TR86-36, Rice University, August 1986.

[14] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[15] David Moon, Richard M. Stallman, and Daniel Weinreb. *Lisp Machine Manual.* MIT, 1983.

[16] Steven P. Reiss. *Displaying Program and Data Structures.* Technical Report CS-86-19, Brown University, April 1986.

[17] Eric J. Golin. Syntax for visual languages. In Progress.

[18] Fred Lakin. Visual grammars for visual languages. In *AAAI87*, pages 683–688, 1987.

Appendix

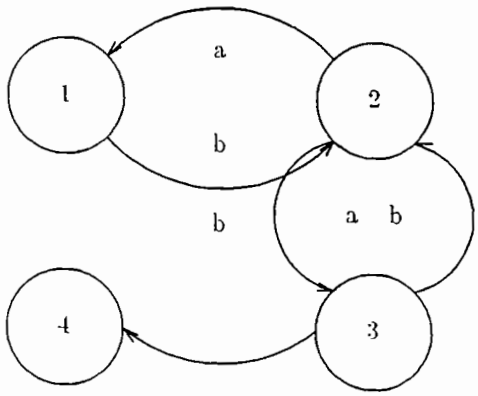

Figure 1: A Sample FSA Program

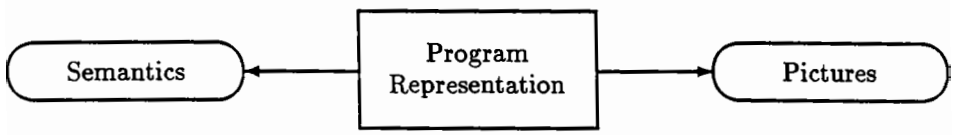

Figure 2: An Environment for Visual Programming

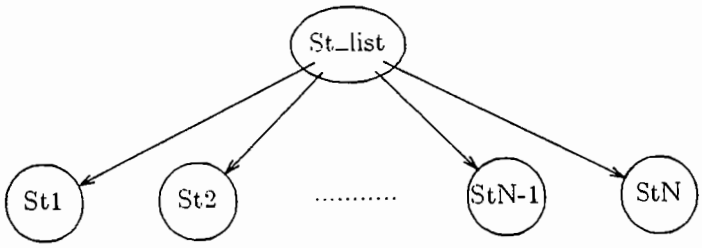

Figure 3: AST Representation of Statement List

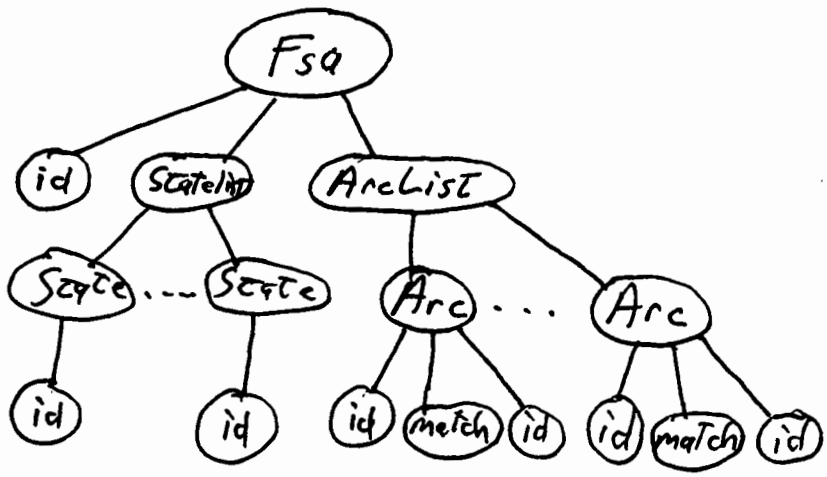

Figure 4: AST for FSA

primitive_og_type[*list of supertypes*] $\Leftarrow$ *domain_name*
list_og_type[*list of supertypes*] $\Leftarrow$ (*list type*)
complex_og_type[*list of supertypes*] $\Leftarrow$ $\langle$*field_name*$_1$:*type*$_1$, *field_name*$_2$:*type*$_2$, ...$\rangle$

Figure 5: Notation for defining Object-Graph Systems

Fsa[] ⇐ ⟨name:String,states:StateList,arcs:ArcList, start:State,current:State⟩
StateList[] ⇐ (State)
ArcList[] ⇐ (Arc)
State[] ⇐ ⟨id:String,accept:Boolean⟩
Arc[] ⇐ ⟨match:Char,from:State,to:State⟩

Figure 6: Object-Graph System for FSA Language

Fsa[] ⇐ ⟨name:String,start:State,current:State⟩
ArcList[] ⇐ (Arc)
State[] ⇐ ⟨id:String,accept:Boolean,next:ArcList⟩
Arc[] ⇐ ⟨match:Char,to:State⟩

Figure 7: Alternate OGS for FSA Language

Block[] ⇐ ⟨body:InitStmt⟩
InitStmt[] ⇐ ⟨next:Statement⟩
Statement[] ⇐ ⟨next:Statement, target:Variable, value:Expression⟩
If[Statement] ⇐ ⟨condition:Expression, then:Statement, else:Statement⟩
Return[Statement] ⇐ ⟨value:Expression⟩

Figure 8: Simple Statement Language

```
Evaluate_Fsa(fsa, inp_ch)
   IF (inp_ch = START_SYMBOL) THEN set fsa.current = fsa.start
                              ELSE set fsa.current = EVAL(fsa.current, inp_ch)
   return (fsa.current).accept

Evaluate_State(state, inp_ch)
   FOR (each arc A in state.next)
      IF EVAL(A, inp_ch) THEN return A.to
   return UNDEFINED$INPUT

Evaluate_Arc(arc, inp_ch)
   IF (inp_ch = arc.match) THEN return TRUE
                           ELSE return FALSE
```

Figure 9: Evaluation Routines for Fsa Language