The Specification of Visual Language Syntax *

Eric J. Golin† Steven P. Reiss

Department of Computer Science, Brown University, Providence, RI 02912

Abstract

Visual programming languages use pictures as programs. A visual programming environment supports the creation of visual programs. We describe an approach to building a language-independent visual programming environment based on syntax specifications. A new model for specifying the syntax of visual languages, picture layout grammars, is described, and an example given of how these grammars are used to define the syntax of a visual language. A spatial parser is an algorithm for recovering the underlying structure of a visual program from the picture. We have implemented a spatial parser for visual languages whose syntax is specified by a picture layout grammar. This parser forms the basis of a visual programming environment.

Introduction

An area of recent research activity has been the development of visual programming languages [1, 2]. Visual programming languages are languages in which programs consist of pictures formed from visual elements as well as text. Much of the research in visual programming has focused on the development of an environment for a specific visual language, generally using an ad-hoc, special purpose interface for creating visual programs.

Part of the reason for this focus is the lack of formal models for specifying visual language syntax and tools for utilizing those specifications. For traditional textual programming languages, formal models such as context-free grammars provide a syntactic specification mechanism; and tools such as *yacc* [3] allow the easy creation of parsers for new languages. Previously, similar tools have not been available for visual languages. Recently, however, several attempts have been made to provide mechanisms for specifying and manipulating visual syntax [4, 5, 6].

Our goal is to develop a language-independent environment for visual programming. Our approach is based on ***picture layout grammars***. Picture layout grammars are a new type of grammar which can be used to specify the syntax of visual languages in much the same way that context-free grammars are used to define textual programming languages. Picture layout grammars are based on a new model of grammar which generates sets of objects with attributes.

We envision an environment where the visual language designer defines his visual language by writing a picture layout grammar. The visual programmer then creates his visual program using a general purpose graphics editor. The visual program (i.e. picture) is then parsed to recover the underlying syntactic structure of the program. This approach has several advantages over a special-purpose environment:

- Using a grammar to specify a visual language provides a precise definition of the syntax of the language.
- Using a parser to recover the language-dependent structure of the program allows the program to be manipulated as a picture. This is analogous to the situation for textual programming languages, where the program may be manipulated as a simple text file. A distinction is maintained between the external representation of the program utilized by the programmer and the internal representation used by semantic components of the environment.
- Because the parser is grammar-driven, creating a parser for a new visual language merely requires defining a new grammar. This forms the basis of a language-independent visual programming environment.

Figure 1 gives the overall architecture of a visual programming system based on a spatial parser. A program consists of a flat collection of graphical objects, such as shapes, lines and text strings. These graphical objects form the primitive elements (i.e. terminal symbols) of a visual language. A picture is created using a graphical editor similar to MacDraw. The spatial parser uses the grammar definition of the visual language to parse the picture and recover the underlying structure of the program.

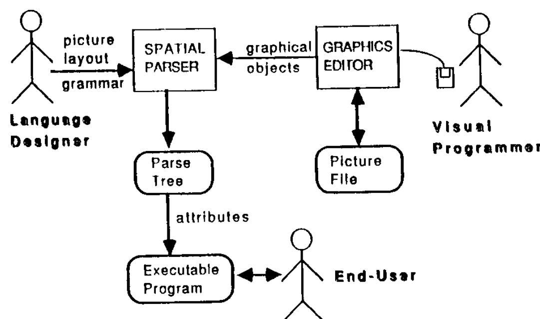

Figure 1: A Visual Programming System based on Spatial Parsing

This paper describes the picture layout grammar mechanism for visual language syntax specification. Section two discusses what comprises a visual language. Section three gives a more formal definition and describes the grammar model used to specify visual languages. Section four gives an example of how picture layout grammars are used to specify a visual language. Section six summarizes our results and relates them to other work.

*This research was supported in part by grants from Defense Advanced Research Projects Agency, the National Science Foundation, and the Digital Equipment Corporation. Equipment support was provided by the National Science Foundation.

†Support for this author was provided by an I.B.M. Graduate Fellowship and by GTE Laboratories. Author's current address: Dept. of Computer Science, University of Illinois, 1304 West Springfield Ave., Urbana, IL 61801

TH0277-4/89/0000/0105$1.00 © 1989 IEEE

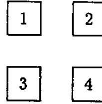

Figure 2: A Two-dimensional Collection of Objects

Visual Languages

This section discusses what is a visual language, what types of visual languages we are interested in, and how visual programs are characterized. The term ***visual language*** is used to describe several types of languages: languages manipulating visual information; languages for supporting visual interactions, and languages for programming with visual expressions [2]. The term ***visual programming languages*** generally refers to the third category. Visual programming languages may be further classified into three groups, according to the extent of visual expression used [7]: icon-based languages, form-based languages and diagram languages.

Visual programming languages form programs from pictures. Languages differ in what kind of pictures they contain, but are similar in the underlying notion of what constitutes a picture. At the lowest level a picture may be viewed as a collection of *primitive elements*, such as lines, polygons and text strings. This is analogous to viewing a textual program as a string of lexical elements. [1]

The primitive elements of a picture may be grouped together into aggregate shapes, which may be further combined to form larger shapes until the entire picture is created. For a textual program, this combination groups adjacent symbols into sub-phrases. The composition of two textual phrases is the concatenation of the two phrases. For a picture, the composition is more complicated than concatenation. The types of compositions may be classified into two groups:

- The composition of adjacent shapes, either fully defined, as in *A above B and the bottom of A touching the top of B*, or less specific, as in *A is next to B.*
- The connection of elements, such as two line segments with a common endpoint.

Using two-dimensional composition operators, a picture may be decomposed into sub-pictures, down to the primitive elements. The distinctions between visual languages arise from both the choice of primitive elements and the composition operators used.

Visual programming languages and textual programming languages are similar in that programs are formed from a basic alphabet and may be decomposed into a well-defined structure. They differ in two important aspects, however. Text strings are one-dimensional and therefore have a linear ordering. A picture, on the other hand, is a two-dimensional object. A linear ordering of the components of a picture would not preserve the adjacency relationship between elements. For example, consider Figure 2 which depicts a picture formed from four boxes. The structure of this picture could have either the boxes labeled 1 and 2 or the boxes labeled 1 and 3 related, depending on the definition of the language. The elements of the picture cannot be simply linearized.

Another important difference between pictures and text is the underlying structure. Text is structured as a tree. In a picture, a sub-shape may compose with several other shapes in a picture. Thus, the underlying structure in a visual language is often a directed graph rather than a tree [8].

Our method of specifying visual language syntax is grammar based. The language implementor specifies the syntax using a *picture layout grammar.* The terminal symbols of the grammar will correspond to the picture elements. The productions correspond to the composition operators of the pictures. A derivation tree (DAG) represents the structure of the picture, with the non-terminal nodes corresponding to the intermediate components.

[1]Both visual and textual languages may be viewed at a lower level - a picture is simply an array of pixels and a text program is merely a string of characters. We are not concerned with the issue of *lexical processing*, or mapping from this lowest level into the elements of syntax

Picture Layout Grammars

This section gives a more precise definition of visual languages and describes our specification method. A textual languages may be viewed as a (possibly infinite) set of strings. Similarly, a picture language may be viewed as a set of pictures, giving us the following definition:

Definition 1 *A* picture element *is a primitive graphical object such as a line, shape or text string. A* picture *is a collection of picture elements arranged on a plane. A* visual language *is a set of pictures. The syntax of a visual language is specified by distinguishing the set of pictures which form the language.*

Using a grammar to characterize a set of pictures (or strings) serves two purposes: it provides a finite definition for an infinite set; and the grammar gives a structure to the elements of the set which forms the basis for parsing.

Picture layout grammars are based on a new grammar model, the ***attributed multiset grammar (AMG)*** [4]. Multiset grammars are similar to context-free grammars, except that the right hand sides of a production are considered to be an unordered collection of symbols, rather than a string. The language generated by a multiset grammar is a set of multisets.[2] An attributed multiset grammar is a multiset grammar which has been augmented with parsing attributes. A production in an attributed multiset grammar is a triple (R, S, C), where

- R is a rewrite rule $N \rightarrow M$, where N is a non-terminal symbol and M is a multiset of symbols.
- S is a set of semantic functions, which map from the attributes of the RHS to the attributes of the LHS.
- C is a set of constraints defined over the attributes of the RHS which indicate when the production is valid.

Attributed multiset grammars are similar to traditional attribute grammars,[9] but differ in several key respects. First, AMG's are based on multiset grammars rather than context-free grammars, so the right-hand side of an AMG production is considered to be unordered. An element of a language recognized by an attributed multiset grammar is an unordered collection of attributed objects. This corresponds to the definition of visual languages given above.

Another difference is that the attributes in an AMG are an integral part of the parsing of an input multiset. The parsing attributes are restricted to synthesized attributes, and the terminal symbols may have synthesized attributes. The attributes influence the parsing through the constraints associated with productions. A parse of an input multiset is valid only if all the constraints are satisfied.

Attributed multiset grammars remove the notion of ordering implicit in a string grammar. Instead, an AMG derivation represents the logical structure of a program. For a visual program, this abstract structure corresponds to the decomposition of the picture into subshapes. A ***picture layout grammar (PLG)*** is an attributed multiset grammar where the productions correspond to picture composition operators. The attributes in a PLG represent the spatial information for a picture element (or aggregate). Each grammar symbol has four attribute lx, by, rx, ty which describe either the area (rectangular extent) occupied by the symbol or the two endpoints $((lx, by)(rx, ty))$ of the symbol. Additional attributes can specify other significant information such as line or fill styles, or the string value of a text object. The constraints in a production specify the relationship between the components of a particular composition. The semantic functions compute the information for the aggregate object.

[2]A *multiset* is a set which may have repeated elements.

over(B,C)	B is over C
left_of(B,C)	B is to the left of C
tiling(B,C,...)	an arbitrary tiling
contains(B,C)	B contains C
group_of(B)	an arbitrary number of B's
adjacent_to(B,C)	B is adjacent to (any direction) C
touches_L(B,C)	(lx, by) of B on the boundary of C
touches_R(B,C)	(rx, ty) of B on the boundary of C
points_from(B,C)	the left end of B is on C
points_to(B,C)	the right end of B is on C
labels(B,C)	B is adjacent to the line C
follow(B,C)	the right end of B is the left end of C
join(B,C)	the right end of B is the right end of C
fork(B,C)	the left end of B is the left end of C
parallel(B,C)	both ends of B and C match
reverse(B)	exchange the left and right ends

Figure 3: Picture Layout Grammar Production Operators

We provide a set of predefined *production operators* which correspond to commonly used compositions. A production operator is defined by a constraint and semantic function which implement the associated composition. A production with a predefined operator is specified as A $\rightarrow op(B, C)$. This is equivalent to a production

$$\begin{array}{ll} A \rightarrow \{B, C\} & \\ \quad A.attr = func_{op}(B.attr, C.attr) & \text{(semantic function)} \\ \textbf{Where:} & \\ \quad pred_{op}(B.attr, C.attr) & \text{(constraint)} \end{array}$$

where $pred_{op}$ specifies the relationship for the composition and $func_{op}$ computes the attributes of the aggregate object.

The production operators defined for picture layout grammars are shown in Figure 3. An example of a production with a predefined operator is A $\rightarrow$ over(B,C), corresponding to the situation show in Figure 4. The operator *over* has the constraint $B.by \geq C.ty$, which ensures that B is located above C. The semantic function for *over* gives A the extent enclosing both B and C.

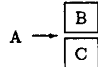

Figure 4: Production A $\rightarrow$ over(B,C)

In addition to the predefined semantic and constraint functions, additional functions and/or constraints can be specified. These additional functions can be used to refine the composition operations, to define new composition operations and to disambiguate the parse. For example, the production in Figure 4 can be rewritten as

$$\begin{array}{l} A \rightarrow \text{over}(B,C) \\ \textbf{Where:} \\ \quad B.by == C.ty \end{array}$$

to specify that B must be over and exactly touching C.

Finally, to allow picture layout grammars to describe directed graph structures, the following extension is made. In a production of a picture layout grammar, an element of the right-hand side may be marked as *remote*. A remote symbol represents a symbol which is not actually part of the production, but must be present elsewhere in the parse tree. This corresponds to introducing a non-tree edge into the parse structure. These edges together with the parse tree form a directed graph. The restriction is made that the graph must be acyclic.

A remote symbol in a production is denoted by underlining the symbol in the right-hand side. The semantic functions and constraints (both the implicit functions associated with a predefined operator and explicit user-specified functions) may refer to a remote symbol in the normal fashion (which serves to determine exactly which node in the parse tree is being referenced). Remote symbols allow a sub-shape to be referred to by (i.e composed with) more than one other sub-shape.

Defining a Visual Language

In this section we give an example of how picture layout grammars are used to specify the syntax of a visual language. The language defined is based on the StateCharts language[10].[3] StateCharts are an extension of finite state automata. StateCharts are interesting because they were developed as a visual programming language and are not based on any textual language. They exploit the two-dimensional nature of pictures to express concurrency in a natural fashion. StateCharts have been used for programming real-time reactive systems, such as embedded control systems.

The StateChart model is based on finite state automata. A finite state automaton (FSA) consists of a set of states and a set of transitions. Each state is identified by a label. A transition may be thought of as a triple $< s_0, e, s_f >$, meaning tate in state s_0, if the next input is event e, move to state s_f. One state in the FSA is designated as the initial state, and one or more states are designated as final states. Finite state automata also form a visual language (e.g. by drawing states as circles and transitions as labelled arcs).

StateCharts enrich finite state automata both semantically and syntactically (i.e. visually). The basic model of computation is similar to that of finite state automata. A StateChart has a current state which changes in response to input events. StateCharts enhance the notion of state with two types of structure: depth and orthogonality. We will now develop the StateChart language and show how it can be specified with a picture layout grammar.

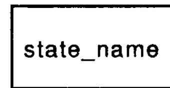

Figure 5: Atomic State

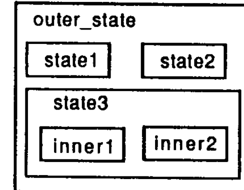

Figure 6: XOR-union of States

The basic entity in a StateChart is a state. The simplest version of a state is a rectangle containing a label, as shown in Figure 5. This is a simple atomic state, similar to a state in an FSA. An atomic state may also represent a more complex state which is left unspecified. StateCharts extend atomic states with *depth*. A state can be formed from the union of a group of states. The containing state is the exclusive-OR (XOR) of the states within it. An FSA consists of a single XOR-union of atomic states. StateCharts extend this notion to allow states to be defined as unions to arbitrary depth.

This hierarchy of states is shown graphically by nesting the group of states within the containing state. Figure 6 shows a StateChart consisting of a state labelled *outer_state* which contains the XOR of three states: *state1*, *state2* and *state3*. *state3* in turn consists of the states *inner1* and *inner2*. The only spatial relationship required between states in an XOR-union is that they are non-overlapping. The label for the containing state must be above all the states in the XOR-union.

Figure 7 gives a picture layout grammar for this part of StateCharts. Nonterminal symbols are shown in CAPITAL letters and terminal symbols in lowercase **bold** letters. Production one says that a STATE is

[3] We have simplified and present only a subset of the language defined by Harel.

(2) STATE → contains(**rectangle**,STATE_INSIDE)
(3) STATE_INSIDE → **text**
(4) STATE_INSIDE → over(**text**,XOR_UNION)
(5) XOR_UNION → STATE
(6) XOR_UNION → adjacent_to(XOR_UNION_1,XOR_UNION_2)

Figure 7: Productions for simple and union states

surrounded by a rectangle. Production two defines a simple atomic state. Productions three through five define an XOR-union. Production five combines the states in the union. This grammar does not necessarily specify a unique decomposition of the XOR-union, but since we are only interested in the group, we don't care how it is decomposed.

The second extension made by StateCharts is to allow a state to be decomposed into orthogonal components. This composition corresponds to the Cartesian product of states. When the StateChart is in the product state, it is also simultaneously in each of the contained states. Just as the previous grouping can be viewed as the OR of states, this product can be seen as the AND of a group of states.

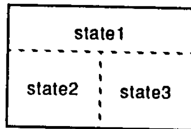

Figure 8: Orthogonal Product

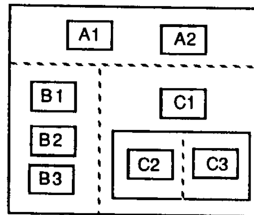

Figure 9: A Complex State

The visual notation for the orthogonal product is to divide the containing state by dashed lines, either vertically or horizontally. Figure 8 shows an example of an orthogonal product of states. When in this state, the machine is simultaneously in *state1*, *state2* and *state3*. Note that no label is given to the product state. Orthogonal products can be combined with XOR-unions, as shown in Figure 9. The state defined by Figure 9 is an element of $(A1 \cup A2) \times (B1 \cup B2 \cup B3) \times (C1 \cup (C2 \times C3))$.

(7) STATE_INSIDE → left_of($STATE_INSIDE_1$,RIGHT_INSIDE)
(8) RIGHT_INSIDE → left_of(VERT_BAR,STATE_INSIDE)
(9) STATE_INSIDE → over($STATE_INSIDE_1$,BOTTOM_INSIDE)
(10) BOTTOM_INSIDE → over(HORIZ_BAR,STATE_INSIDE)
(11) VERT_BAR → **line**
Where
line.LX == line.RX
line.style == DASHED_STYLE
(12) HORIZ_BAR → **line**
Where
line.BY == line.TY
line.style == DASHED_STYLE

Figure 10: Productions for orthogonal product of states

Picture layout grammar productions for the orthogonal product of states are given in Figure 10. Productions six through nine build up a state from horizontal and vertical compositions of states. Production ten (eleven) uses additional attributes to specify exactly what forms a vertical (horizontal) bar.

Similar to finite state automata, StateCharts have transitions defined on input events. The visual notation for a transition is a labelled arc between states, as shown in Figure 11. The label is a text string specifying the input event. Possible PLG productions for a transition are

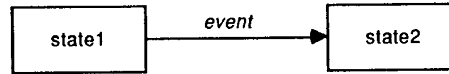

Figure 11: A Transition

LABELLED_ARROW → labels(**text**,**arrow**)
LABELLED_ARC → points_to(LABELLED_ARROW,<u>STATE</u>)

The first production uses the ***labels*** operator to relate an arrow with the text string labelling it. The second production defines a LABELLED_ARC to be an arrow which points to a STATE. Note that the STATE operand in the second production is underlined. This indicates that STATE is a *remote* argument, so the STATE is not actually below the LABELLED_ARC in the parse tree. The remote STATE operand serves two purposes: It expresses the syntactic construction that a transition must point to a destination state and it relates the transition to that destination state with a non-tree edge in the parse structure.

Now we need to relate the transitions to the state which they are leaving. We can do this with a production such as STATE → touches_L(LABELLED_ARC,$STATE_1$), which combines all the arcs leaving a state with that state one at a time. As with the XOR-union, this production does not specify a unique parse structure. First of all, no order is given for the arcs to combine with the state. This is not a problem for *expansive* productions (productions where the LHS has a larger extent than the individual elements of the RHS), but the *touches_L* operator is not expansive. We need to disambiguate this production by introducing an ordering on the arcs. Since it is only important that there is an ordering, we can use an arbitrary one. A second problem is that we would like any incoming arcs to point to the bottommost STATE in the parse tree (i.e. the state formed by the rectangle). Both of these problems can be solved by introducing a new nonterminal symbol *STATE0* and a new attribute *pos*, as shown in Figure 12 below.

(13) STATE0 → contains(**rectangle**,STATE_INSIDE)
(14) STATE → STATE0
STATE.pos = 0
(15) STATE → touches_L(LABELLED_ARC,$STATE_1$)
STATE.pos = hash(LABELLED_ARC)
Where
$STATE_1$.pos < hash(LABELLED_ARC)
(16) LABELLED_ARC → points_to(LABELLED_ARROW,<u>STATE0</u>)
(17) LABELLED_ARROW → labels(**text**,**arrow**)

Figure 12: Transition productions

Here production 12 replaces production one above. The ***hash*** function maps from the extent of a symbol to a unique positive integer, which serves as the arbitrary ordering of the arcs around a STATE. The *pos* attribute is used to implement the ordering. Production 15 fixes the LABELLED_ARC to point to the STATE0 object.

Finally, StateCharts make two extensions to allow transitions to work with the hierarchy. For example, consider the statechart shown in Figure 13, which has a transition from state A to state B on event $E1$. Since state B is the XOR-union of states B_1 and B_2, we must specify which of these states is to be entered. The first extension is the use of ***default states***. The default state specifies which state is to be entered when a group is entered. It is indicated visually by a unlabelled arrow starting at an circle containing the letter 'D' and pointing to the default state. A default state is analogous to the start state of a finite state automata.

The second extension is to add memory to an XOR-union. A circle containing the letter 'H' represents the most recently visited state. A transition ending at the history symbol signifies a transition to what-

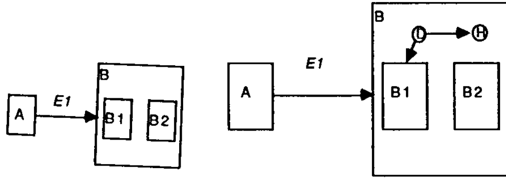

Figure 13: Underspecified Transition

Figure 14: Default and History

ever state was most recently visited. Figure 14 shows a StateChart with both history and default states. The default state has two arrows - one to the history and the other to state $B1$. This signifies that the default state is initially state $B1$ and then is the most recently visited state.

(18) DEFAULT → DEF_STATE
(19) DEFAULT → touches_L(UNLBLD_ARC_HIST,DEF_STATE)
(20) DEF_STATE → touches_L(UNLBLD_ARC_STATE,DEF_SYMBOL)
(21) DEF_SYMBOL → contains(**circle**,**text**)
Where
text.sval == "D"
(22) HISTORY → contains(**circle**,**text**)
Where
text.sval == "H"
(23) UNLBLD_ARC_HIST → points_to(**arrow**,HISTORY)
(24) UNLBLD_ARC_STATE → points_to(**arrow**,STATE0)
(25) XOR_GROUP → STATE
XOR_GROUP.default = 0
XOR_GROUP.history = 0
(26) XOR_GROUP → DEFAULT
XOR_GROUP.default = 1
XOR_GROUP.history = 0
(27) XOR_GROUP → HISTORY
XOR_GROUP.default = 0
XOR_GROUP.history = 1
(28) XOR_GROUP → adjacent_to(XOR_GROUP_1,XOR_GROUP_2)
XOR_GROUP.default = XOR_GROUP_1.default + XOR_GROUP_2.default
XOR_GROUP.history = XOR_GROUP_1.history + XOR_GROUP_2.history
Where
$XOR_GROUP_1.default + XOR_GROUP_2.default \leq 1$
$XOR_GROUP_1.history + XOR_GROUP_2.history \leq 1$

Figure 15: Default and History Productions

The PLG productions for default states and history are given in Figure 15. A DEFAULT is specified as an **circle** containing a 'D' with an unlabelled **arrow** pointing to a STATE0 and an optional unlabelled **arrow** pointing to a HISTORY. A HISTORY is specified as an **circle** containing an 'H'. Productions 24 through 27 are used to include the default and history symbols into the XOR-union and replace productions four and five. Two additional attributes, *default* and *history*, are used to ensure that an XOR-union has at most one history and default symbol.

We complete the specification of our StateChart language with the production,

CHART → STATE

which defines a StateChart to be a STATE. Figure 16 shows an example of StateChart. The parse structure for this StateChart is given in Figure 17. The interior nodes of the parse tree are labelled with nonterminal symbols. The leaf nodes are labelled by the terminal symbols. Note the thick grey arrow from the LABELLED_ARC node to the STATE0 node. This indicates that STATE0 was a remote symbol in the LABELLED_ARC production.

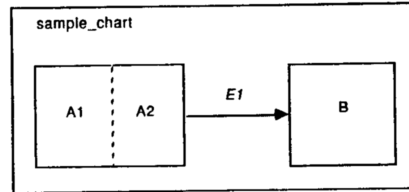

Figure 16: A Sample StateChart

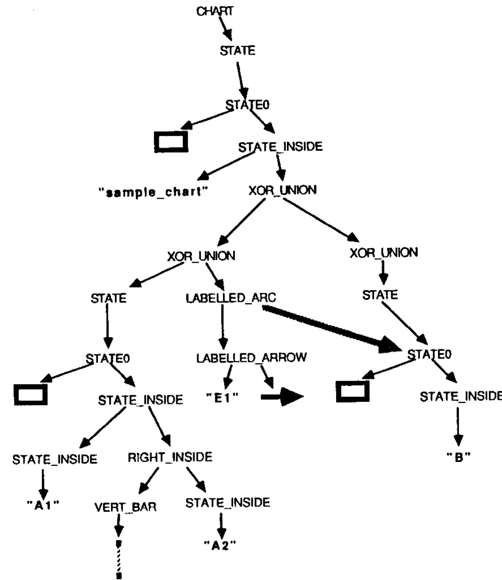

Figure 17: Parse structure for StateChart

The spatial parser takes as input the ten terminal symbols (three rectangles, five text strings, a line and an arrow) shown in Figure 16. Each terminal symbol has attributes which express its location, line style, etc. The parser processes the terminal symbols and produces the parse structure shown in Figure 17. The spatial parsing algorithm is described in [4].

In addition to the attributes used for parsing, the language designer can specify functions to compute semantic attributes, which may be both syntheisized and inherited. These attributes may be used to compute values, take actions, compile the program or translate the program into an abstract representation.

Conclusion

This paper describes a mechanism for defining the syntax of two dimensional languages. We have implemented a visual programming environment based on picture layout grammars. The environment combines a graphics editor with a spatial parser. The language designer creates a text file containing a picture layout grammar, in a format similar to a *yacc* input grammar. The visual programmer interacts with the editor to create a picture (i.e. program). The grammar can be loaded and

the picture parsed. External functions may be dynamically loaded and used in attribute expressions. The environment is implemented in C, using the X Window System and the Brown Workstation Environment.

Several other researchers have looked at the problem of parsing pictures. Lakin [6] coined the term *spatial parsing* and described a type of grammar for specifying visual languages. He advocated using a general purpose graphics editor to create a picture and recovering its structure with a parser. He did not give a formal model for his visually annotated grammars or an efficient general parser.

The SIL project has also attempted to build a programming environment using a spatial parser [11]. The SIL compiler is designed to handle icon-oriented languages. It uses a *picture grammar*, which is a context-free grammar augmented with spatial operators. The syntax analysis is performed by first transforming the picture into a pattern string and then using a lookahead LR parser. The spatial operators are limited to horizontal or vertical concatenation and spatial overlay. Because the picture must first be transformed into a pattern string, the SIL syntax mechanism does not extend to complex visual languages such as Statecharts [10]. The SIL project demonstrates the utility of a parser for a visual environment.

Other attempts at using grammars to describe pictures can be found in the research on syntactic pattern recognition [12]. Shaw's *Picture Description Language* used a context free grammar augmented by concatenation operators to describe a picture [13]. Bunke has described the use of *attributed programmed graph grammars* for interpreting diagrams.[14] Fu has suggested the use of an attribute grammar for picture recognition [15]. Syntactic pattern recognition applications differ from our goal in that the pictures are not actually language elements, and the grammars are used to capture the structure of the pattern.

We have described a specification model for visual language syntax. Our model has proven to widely applicable. We have defined a number of visual languages using picture layout grammars, all of which can then be parsed within our visual programming environment. The grammar model is easily extended to new visual compositions by providing new semantic functions and constraints. The parser has also been used within a visual programming environment developed at GTE Laboratories [16]. Our continuing research is concentrated on two issues: better integration of semantic processing and investigation of other other tools based on picture layout grammars.

References

[1] S. K. Chang, "Visual languages: A tutorial and survey," *IEEE Software*, vol. 4, pp. 29–39, Jan. 1987.

[2] N. C. Shu, *Visual Programming*. Van Nostrand Reinhold Company, 1988.

[3] S. C. Johnson, "Yacc: Yet another compiler-compiler," tech. rep., Bell Laboratories, 1974.

[4] E. J. Golin, *A Method for the Specification and Parsing of Visual Languages*. PhD thesis, Brown University, 1989.

[5] G. Tortora and P. Leoncini, "A model for the specification and interpretation of visual languages," in *1988 IEEE Workshop on Visual Languages*, pp. 52–60, IEEE, Oct. 1988.

[6] F. Lakin, "Spatial parsing for visual languages," in *Visual Languages* (S.-K. Chang, T. Ichikawa, and P. A. Ligomenides, eds.), pp. 35–85, Plenum Press, 1986.

[7] N. C. Shu, "Visual programming languages: A perspective and a dimensional analysis," in *Visual Languages* (S.-K. Chang, T. Ichikawa, and P. A. Ligomenides, eds.), pp. 11–34, Plenum Press, 1986.

[8] E. J. Golin and S. P. Reiss, "Representing visual programs with object graphs," Tech. Rep. CS-89-05, Brown University, 1987.

[9] D. E. Knuth, "Semantics of context-free languages," *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127–145, 1968.

[10] D. Harel, "Statecharts: A visual approach to complex systems," Tech. Rep. CS-84-05, Weizmann Institue of Science, July 1984.

[11] S.-K. Chang, "The design of a visual language compiler," in *1988 IEEE Workshop on Visual Languages*, pp. 84–90, IEEE, Oct. 1988.

[12] K. S. Fu, *Syntactic Pattern Recognition and Applications*. Prentice-Hall, Inc., 1982.

[13] A. C. Shaw, "A formal picture description scheme as a basis for picture processing systems," *Information and Control*, vol. 14, pp. 9–52, 1969.

[14] H. Bunke, "Attributed programmed graph grammars and their application to schematic diagram interpretation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-4, pp. 574–582, Nov. 1982.

[15] K. C. You and K.-S. Fu, "A syntactic approach to shape recognition using attributed grammars," *IEEE Trans. on Systems, Man and Cybernetics*, vol. SMC-9, pp. 334–345, June 1979.

[16] E. Golin, R. V. Rubin, and J. W. II, "The visual programmers workbench," in *11th World Computer Congress*, IFIP, Aug. 1989.