Journal of Visual Languages and Computing (1990) 1, 141–157

The Specification of Visual Language Syntax*

ERIC J. GOLIN† AND STEVEN P. REISS

Brown University, Department of Computer Science, Providence, RI 02912, U.S.A.

Visual programming languages use pictures as programs. One approach to building a visual programming environment is to parameterize a generic environment with a language specification. We describe a mechanism for specifying visual languages that can be used as the basis of a language-independent visual programming environment. Our mechanism is a new type of grammar, called a picture layout grammar. We show how this type of grammar can describe the two-dimensional syntax of a visual language and give an example of its use. A picture layout grammar permits the syntactic structure of visual program to be recovered by parsing. The parsing ability provides the basis of our visual programming environment.

1. Introduction

AN AREA OF RECENT RESEARCH ACTIVITY has been the development of visual programming languages [1, 2]. Visual programming languages use pictures to express computations. Visual programming languages exploit both the graphical interface and powerful processors found in state-of-the-art workstations. A pictorial presentation is often far more expressive than a textural representation of the same information. A finite state automaton could be described using text or a program, but a finite state diagram is much easier to understand.

Much of the research in visual programming has focused on the development of specific visual languages. For example, visual languages have been implemented for functional programming [3], icon based flowcharts [4], programming by demonstration [5], finite state automata [6], and data flow [7]. Implementation of these languages has required the construction of *ad hoc* special purpose environments for creating, manipulating and processing visual programs. As a result, development of a new visual programming environment requires substantial effort.

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' [8] allow the easy creation of parsers for new languages. Similar tools have not been available for visual languages until recently, when several attempts have been made to provide mechanisms for specifying and manipulating visual syntax [9–11].

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

† Support for this author was provided by an I.B.M. Graduate Fellowship. Author's current address is Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield Avenue, Urbana, IL 61801, U.S.A.

1045-926X/90/010141 + 17 $03.00/0

© 1990 Academic Press Limited

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. Processing of the program (e.g. compilation) can be performed using the extracted structure. Figure 1 gives the overall architecture of such a visual programming system.

The graphics editor provides the user with the ability to interactively create and modify pictures by placing graphical objects such as shapes, lines and text strings on the screen. These graphical objects form the primitive elements (i.e. terminal symbols) of a visual language. The editor maintains and manipulates its own representation of the picture. This is analogous to the situation with conventional programming languages, where a program is constructed using a generic text editor. In both cases, the programmer manipulates a source file.

The spatial parser uses the grammar definition of the visual language to parse the picture and recover the underlying structure of the program. The parser maps from the flat representation of the picture as a collection of graphical elements to a parse tree. The parse tree is augmented with attributes, which can be used to provide semantic processing for a visual language.

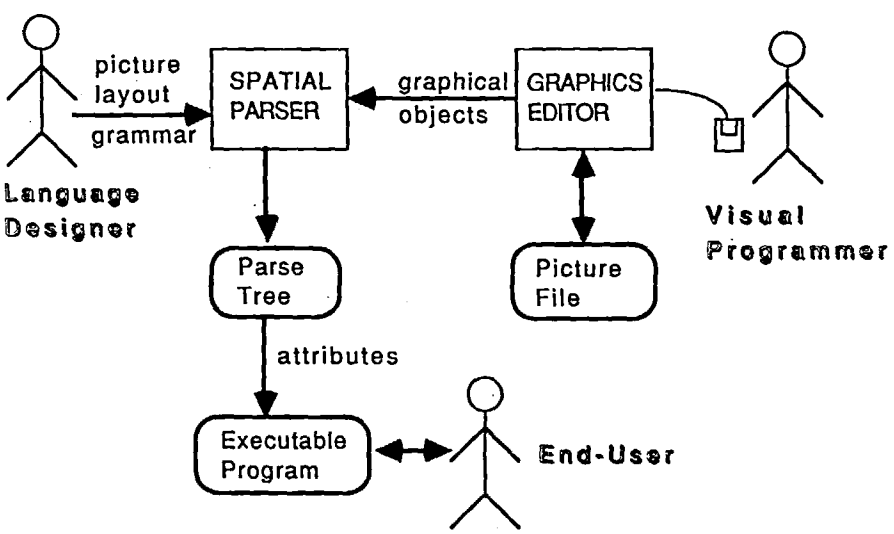

Figure 1. A visual programming system based on spatial parsing

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. Thus, the same editor may be used for different visual languages.
- Since the parser is grammar-driven, a parser for a new visual language is created by merely writing a new grammar. The effort in developing a new visual language is greatly reduced.

This paper describes the picture layout grammar mechanism for visual language syntax specification. Section 2 discusses what comprises a visual language. Section 3 gives a more formal definition of visual languages and describes the grammar model used to specify visual languages. Section 4 gives an example of how picture layout grammars are used to specify a visual language. Section 5 describes our implementation and Section 6 summarizes our results and relates them to other work.

2. Visual Languages

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, according to the type and extent of visual expression used, into icon-based languages, form-based languages and diagram languages [12].

Figure 2 shows example programs from three visual languages: a finite state diagram

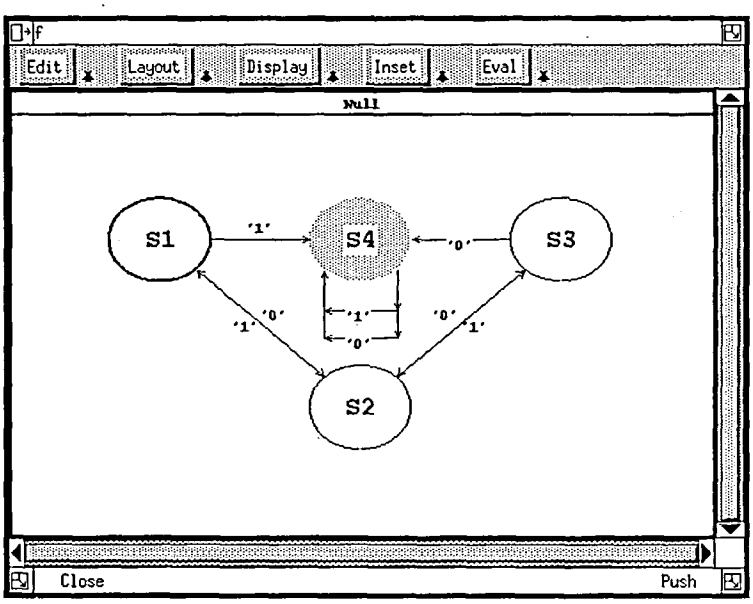

Figure 2. Visual language examples. (a) A GARDEN fsa program [13]

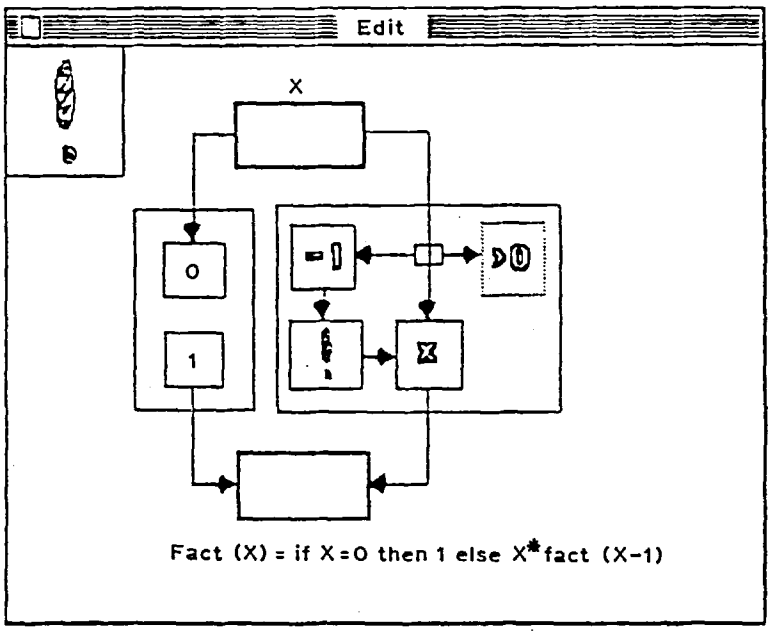

Figure 2. (b) A Show and Tell program redrawn by permission of T. D. Kimura [7]

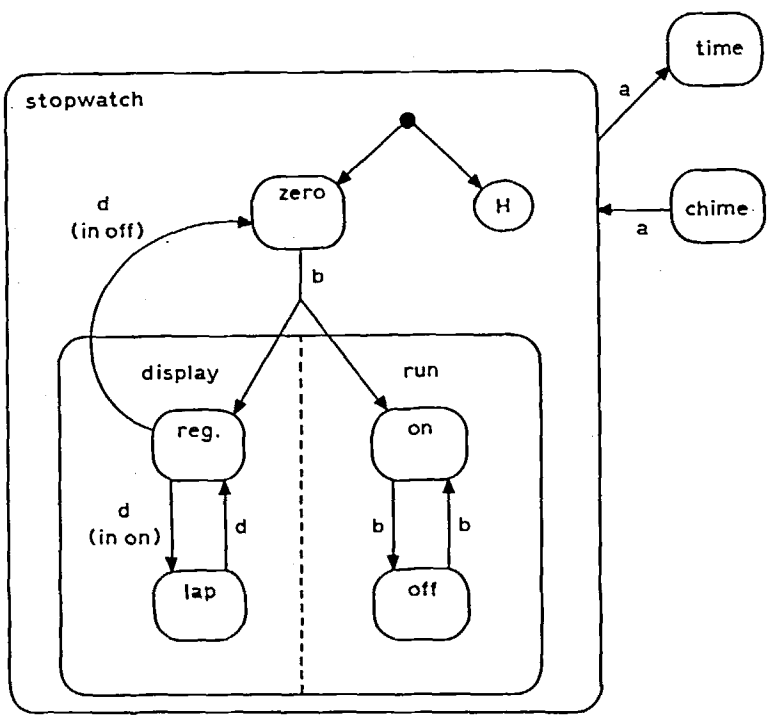

Figure 2. (c) A StateChart program [14]

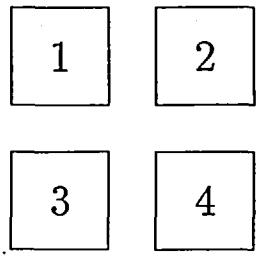

Figure 3. A two-dimensional collection of objects

developed with the GARDEN system [13]; a Show and Tell puzzle [7]; and a StateChart program [14]. These languages all use pictures to form programs. The languages differ syntactically, in that the set of pictures constituting valid programs is different for each language. (They also differ semantically, in the meaning that is assigned to programs.) They are similar, however, 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.[a]

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 subphrases. The composition of two textual phrases is their concatenation. 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 syntactic distinctions between the visual languages in Figure 2 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 over 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 3 which depicts a picture formed from four boxes. The structure of this picture could have either the boxes labelled 1 and 2, or the boxes labelled 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.

[a] 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.

The essentially linear nature of text is reflected in an underlying tree structure. In a picture, a subshape may compose with several other shapes, rather than just an immediate left or right neighbour. Thus, the underlying structure in a visual language is often a directed graph rather than a tree [15].

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 (actually a directed acyclic graph) represents the structure of the picture, with the non-terminal nodes corresponding to the intermediate components.

3. Picture Layout Grammars

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

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 defining 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 both parsing and syntax-directed processing.

Picture layout grammars are based on a new grammar model, the attributed multiset grammar (AMG) [11]. 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.[b] 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 semantic function, which maps from the attributes of the right-hand side (RHS) to the attributes of the left-hand side (LHS).
- C is a constraint defined over the attributes of the RHS which indicates when the production is valid.

Attributed multiset grammars are similar to traditional attribute grammars [16], but differ in several key respects. First, AMGs 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

[b] A multiset is a set which may have repeated element, and is what we mean by an unordered collection.

parsing of an input multiset. Only synthesized attributes are permitted for parsing, but 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 grammar (PLG) is an attributed multiset grammar where the production correspond to picture composition operators.

The attributes in a PLG represent the spatial information for a picture element (or aggregate). Each grammer symbol has four attributes *lx*, *by*, *rx*, *ty*. Abstractly, there are two types of primitive elements in picture layout grammars: shapes (where the attributes describe the rectangular extent); and lines [where the attributes give the two endpoints (*lx*, *by*) (*rx*, *ty*) of the symbol]. More specialized graphical primitives are viewed as instances of one of these two types. For example, text objects and ellipses are both considered special types of shape, and arrows are a special type of line. Additional attributes are used to specify other significant information, such as line style 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. 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

$A \rightarrow \{B, C\}$	(the rewrite rule)
$A.attr = func_{op}(B.attr, C.attr)$	(the semantic function)
Where:	
$pred_{op}(B.attr, C.attr)$	(the constraint)

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 4. An example of a production with a predefined operator is $A \rightarrow \text{over}(B, C)$, corresponding to the situation shown in Figure 5. 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.

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 5 can be rewritten as

$$A \rightarrow \text{over}(B, C)$$
Where:
$$B.by == C.ty$$
$$B.lx == C.lx$$
$$B.rx == C.rx$$

to specify that B must be over and exactly touching C, with the left and right edges aligned.

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 area containing an arbitrary number of *B* objects
adjacent_to(*B*, *C*)	*B* is adjacent to (any direction) *C*
touches_L(*B*, *C*)	the left (*lx*, *by*) point of *B* is on the boundary of *C*
touches_R(*B*, *C*)	the right (*rx*, *ty*) point of *B* is 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 4. Picture layout grammar production operators

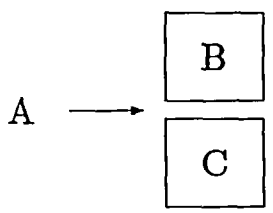

Figure 5. Production $A \rightarrow$ over(B, C)

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 an additional edge into the parse tree. These edges together with the parse tree form a directed graph. The restriction is made that the graph must be acyclic. (Note that this restriction is on the parse structure, and not on the picture.)

A remote symbol in a production is denoted by underlining the symbol in the RHS. 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 subshape to be referred to by (i.e. composed with) more than one other sub-shape.

4. 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 [14][c]. StateCharts are an extension of finite state automata. StateCharts are

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

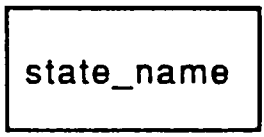

Figure 6. Atomic state

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 $\langle s_0, e, s_f \rangle$, meaning that 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.

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 6. 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 7 shows a StateChart consisting of a state labelled outer_state which contains the XOR of three states: state1, state2 and state3. In turn,

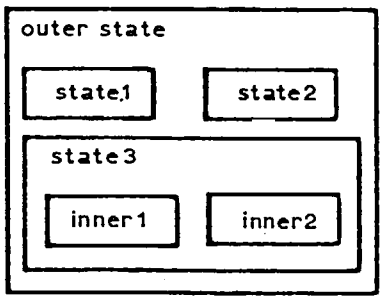

Figure 7. XOR-union of states

(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 8. Productions for simple and union states

state3 consists of the states inner1 and inner2. The only spatial relationship required between states in an XOR-union is that they are nonoverlapping. The label for the containing state must be above all the states in the XOR-union.

Figure 8 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 1 says that a STATE is surrounded by a rectangle. Production 2 defines a simple atomic state. Productions 3 through 5 define an XOR-union. Production 5 combines the states in the union. This grammar does not necessarily specify a unique decomposition of XOR-union, but since we are only interested in the group, we do not 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 XOR of states, this product can be seen as the AND of a group of states.

The visual notation for the orthogonal product is to divide the containing state by dashed lines, either vertically or horizontally. Figure 9 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 10. The state defined by Figure 10 is an element of $(A1 \cup A2) \times (B1 \cup B2 \cup B3) \times (C1 \cup (C2 \times C3))$.

Picture layout grammar productions for the orthogonal product of states are given in Figure 11. Productions 6 through 9 build up a state from horizontal and vertical compositions of states. Production 10 (11) 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 12. The label is a text string specifying the input event. Possible PLG productions for a transition are:

LABELLED_ARROW→ labels(**text**,**arrow**)
LABELLED_ARC→ points_to(LABELLED_ARROW,STATE)

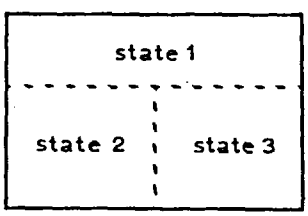

Figure 9. Orthogonal product of states

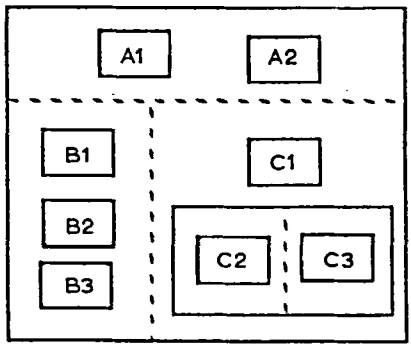

Figure 10. A complex state

```
(7)    STATE_INSIDE→ left_of(STATE_INSIDE1,RIGHT_INSIDE)
(8)    RIGHT_INSIDE→ left_of(VERT_BAR,STATE_INSIDE)
(9)    STATE_INSIDE→ over(STATE_INSIDE1,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 11. Productions for orthogonal product of states

Figure 12. A transition

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

$$\text{STATE} \rightarrow \text{touches\_L(LABELLED\_ARC,STATE}_1\text{)},$$

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, such as over), 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 13 below.

```
(13)   STATE0 → contains(rectangle,STATE_INSIDE)
(14)   STATE → STATE0
                   STATE.pos = 0
(15)   STATE → touches_L(LABELLED_ARC,STATE1)
                   STATE.pos = hash(LABELLED_ARC)
       Where
                   STATE1.pos < hash(LABELLED_ARC)
(16)   LABELLED_ARC → points_to(LABELLED_ARROW,STATE0)
(17)   LABELLED_ARROW → labels(text,arrow)
```

Figure 13. Transition productions

Here production 13 replaces production 1 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 makes two extensions to allow transitions to work with the hierarchy. For example, consider the statechart shown in Figure 14, 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 an unlabelled arrow starting at an circle containing the letter 'D' and pointing to the default state. A default state is analogous to the start 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 whatever state was most recently visited. Figure 15 shows a StateChart with both history and default states. The default state is initially state $B1$ and then is the most recently visited state.

The PLG productions for default states and history are given in Figure 16. A DEFAULT is specified as an circle containing a 'D' with an unlabelled arrow

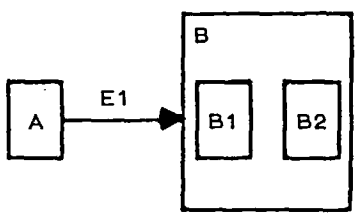

Figure 14. An underspecified transition

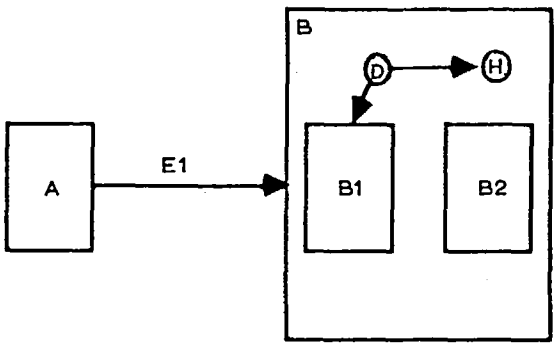

Figure 15. Default and history states

pointing to a STATE0 and an optional unlabelled arrow pointing to a HISTORY. A HISTORY is specified as a circle containing an 'H'. Productions 24 through 27 are used to include the default and history symbols into the XOR-union and replace productions 4 and 5. 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 17 shows an example of StateChart. The parse structure for this StateChart is given in Figure 18. The interior nodes of the

(18) DEFAULT → DEFAULT_STATE
(19) DEFAULT → touches_L(UNLABELLED_ARC_HIST,DEFAULT_STATE)
(20) DEFAULT_STATE → touches_L(UNLABELLED_ARC_STATE,DEFAULT_SYMBOL)
(21) DEFAULT_SYMBOL → contains(**circle,text**)
Where
text.sval == "D"
(22) HISTORY → contains(**circle,text**)
Where
text.sval == "H"
(23) UNLABELLED_ARC_HIST → points_to(**arrow**,HISTORY)
(24) UNLABELLED_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 16. Default and history productions

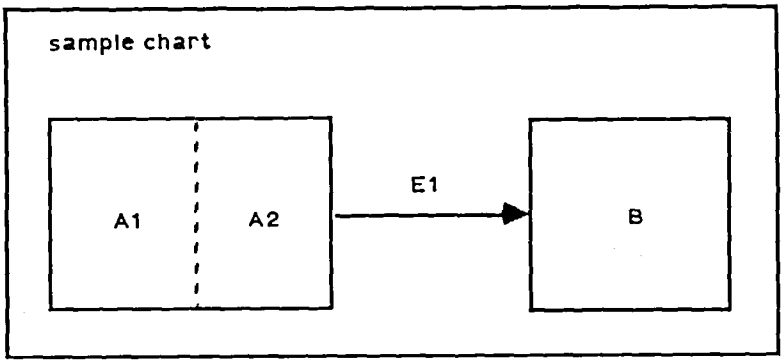

Figure 17. A sample StateChart

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.

The spatial parser takes as input the ten terminal symbols (three rectangles, five text strings, a line and an arrow) shown in Figure 17. Each terminal symbol has attributes

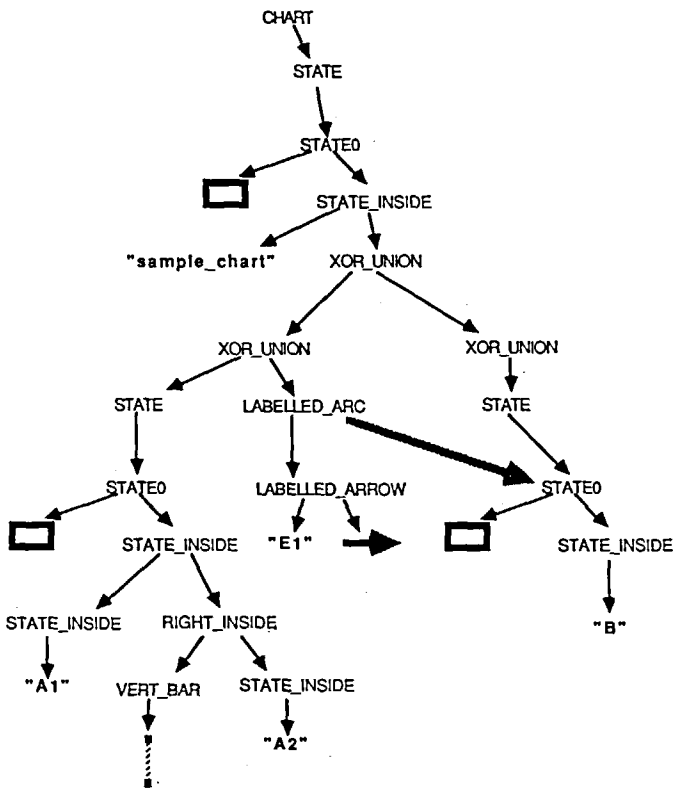

Figure 18. Parse structure for StateChart

which express its location, line style, etc. The parser processes the terminal symbols and produces the parse structure shown in Figure 18. The spatial parsing algorithm is described in [11].

5. The GREEN Environment

The GRaphical Editing ENvironment (or GREEN) is a visual programming environment based on picture layout grammars implemented at Brown University. [11] The environment combines a graphics editor with a grammar driven spatial parser. It allows the definition of visual languages and provides the means to create and process visual programs.

A visual language is specified by a text file containing a picture layout grammar. The file is in a format similar to a yacc input grammar, and contains definitions of the objects in the grammar (e.g. attributes, terminals and nonterminals) and a list of productions. GREEN reads the language specification from the grammar file.

A visual programmer uses the editor to create a picture. The editor is a simple object-based graphics editor. A picture is a collection of primitive graphical objects. The editor currently supports rectangles, octagons, circles, text strings, line segments and arrows as primitve object types. A picture is edited by picking commands from menus and using the mouse for positioning and selection. Pictures may be saved and loaded from files, and several pictures may be edited at once. A grid is provided for aligning the positions of objects.

GREEN contains an implementation of a spatial parsing algorithm for picture layout grammars. The spatial parser uses the language specification read from a grammar file to analyse the current picture. The result of the parse is an attributed parse structure.

The attribute language provides three data types: integers, strings and lists. Operations on these types are provided by builtin functions. New types and additional functions may be declared in the grammar file. For a new function, the declaration gives a name for the function within the attribute expressions as well as the name of an external routine implementing the function. The external routine is dynamically loaded from an object file. The hash function used in Figure 13 is an example of an external function.

In addition to the attributes used for parsing, the language designer can specify processing attributes for symbols. These attributes may be used to perform static checks, compute values, take actions, compile the program or translate the program into an abstract representation. The processing attributes may be either synthesized or inherited. They are evaluated in a separate phase after the parsing is complete. Attribute values of the root node in the parse structure can be accessed from within GREEN. External functions allow the specification of other actions to take following parsing (e.g. writing an object file).

GREEN is implemented in the C programming language, using the X Window System and the Brown Workstation Environment [17]. It runs on workstations from Sun Microsystems (Sun 3, Sun 4 and Sparcstation). Picture layout grammars have been written for a number of visual languages, including Statecharts, finite state diagrams and expression trees.

6. Conclusion

This paper describes a mechanism for defining the syntax of two-dimensional languages. Several other researchers have looked at the problem of specifying visual languages with grammars. Lakin [9] 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 [18]. 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. 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 [19]. Shaw's Picture Description Language used a context free grammar augmented by concatenation operators to describe a picture [20]. Bunke has described the use of attributed programmed graph grammars for interpreting diagrams [21]. Fu has suggested the use of an attribute grammar for picture recognition [22]. 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 be 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 [23]. Our continuing research is concentrated on two issues-better integration of semantic processing and investigation of other tools based on picture layout grammars.

References

1. S. K. Chang (1987) Visual languages: A tutorial and survey. *IEEE Software* 4, 29–39.
2. N. C. Shu (1988) Visual Programming. Van Nostrand Reinhold Company, New York, 315pp.
3. G. Raeder (1984) *Programming in Pictures.* PhD thesis, University of Southern California, 1984.
4. E. P. Glinert & S. L. Tanimoto (1984) Pict: an interactive graphical programming environment, *IEEE Computer* 17, 7–25.
5. R. V. Rubin, E. J. Golin & S. P. Reiss (1985) Thinkpad: A graphical system for programming by demonstration. *IEEE Software* 2, 73–79.
6. R. J. K. Jacob (1985) A state transition diagram language for visual programming. *IEEE Computer* 18, 51–59.
7. T. D. Kimura, J. W. Choi & J. M. Mack (1986) A visual language for keyboardless programming, Technical report WUCS-86-6, Washington University, 52pp.

8. S. C. Johnson (1974) Yacc: Yet another compiler-compiler. Technical report, Bell Laboratories.
9. F. Lakin (1986) Spatial parsing for visual languages. In *Visual Languages* (S.-K. Chang, T. Ichikawa, and P. A. Ligomenides, eds), Plenum Press; New York, pp. 35–85.
10. G. Tortora & P. Leoncini (1988) A model for the specification and interpretation of visual languages. In: *1988 IEEE Workshop on Visual Languages* (Pitsburgh, PA), IEEE, pp. 52–60.
11. E. J. Golin (1990) *A Method for the Specification and Parsing of Visual Languages.* PhD thesis, Brown University.
12. N. C. Shu (1986) Visual programming languages: A perspective and a dimensional analysis. In *Visual Languages* (S.-K. Chang, T. Ichikawa & P. A. Ligomenides, eds), Plenum Press, New York, pp. 11–34.
13. S. P. Reiss (1987) Working in the Garden environment for conceptual programming, *IEEE Software* 4, 16–27.
14. D. Harel (1987) Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8, 231–274.
15. E. J. Golin & S. P. Reiss (1987) Representing visual programs with object graphs, Technical Report CS-89-05, Brown University.
16. D. E. Knuth (1968) Semantics of context-free languages. *Mathematical Systems Theory* 2, 127–145.
17. S. P. Reiss & J. T. Stasko (1989) The Brown Workstation Environment: A user interface design toolkit. In: *IFIP Working Conference on Engineering for Human Computer Communication* (Napa Valley, CA), North Holland, 1989.
18. S.-K. Chang, M. J. Tauber, B. Yu & J.-S. Yu (1989) A visual language compiler. *IEEE Transactions on Software Engineering* 15, 506–525.
19. K. S. Fu (1982) *Syntactic Pattern Recognition and Applications.* Prentice-Hall, Inc., New York, 596pp.
20. A. C. Shaw (1969) A formal picture description scheme as a basis for picture processing systems. *Information and Control* 14, 9–52.
21. H. Bunke (1982) Attributed programmed graph grammars and their application to schematic diagram interpretation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-4, 574–582.
22. K. C. You & K.-S. Fu (1979) A syntactic approach to shape recognition using attributed grammars. *IEEE Transactions on Systems, Man and Cybernetics* SMC-9, 334–345.
23. E. Golin, R. V. Rubin & J. Walker II (1989) The visual programmers workbench, In: *Proceedings of IFIP World Computer Conference* (San Francisco, CA), North-Holland, 1989.