Parsing in a Visual Language Environment

Eric J. Golin
Steven P. Reiss

Brown University
Department of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-89-06

February, 1989

This is a preliminary version, a revised edition of this paper is forthcoming.

Parsing in a Visual Language Environment
Extended Abstract

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

February, 1989

Abstract

Visual programming languages use pictures as programs. A visual programming environment supports the creation of visual programs. A spatial parser recovers the underlying structure of a visual program from the picture. We describe how a spatial parser can be used to form the basis of a language-independent visual programming environment. A new model for specifying the syntax of visual language, the Picture Layout Grammar, is described. Finally, an algorithm for parsing visual programs based on a PLG specification is given.

*Support for this author was provided by an I.B.M. Graduate Fellowship and by GTE Laboratories.

1 Introduction

An area of recent research activity has been the development of visual programming languages [1, 2, 3]. Visual programming languages are languages in which programs consist of pictures formed from visual elements as well as text. Visual languages utilize the power of todays workstations to provide a graphical interface. A pictorial presentation is often far more expressive than a textual representation of the same information. For example, a diagram of a finite state automaton is easier to understand than either a textual description or a program implementing the fsa.

Much of the research in visual programming has focused on the development of an environment for a specific visual language. For example, visual languages have been implemented for functional programming [4], icon based flowcharts [5], programming by demonstration [6], finite state automata [7], and data flow [8]. Most of these systems use an ad-hoc, special purpose interace for creating 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* [9] 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 [10, 11, 12].

Our goal is to develop a language-independent environment for visual programming. Our approach is based on *spatial parsing*.[12, 13] The programmer creates his program using a general purpose graphics editor. The picture is then parsed to recover the underlying structure of the program. The parser is driven by a *picture layout grammar* which specifies the syntax of the language. This approach has several advantages over a special-purpose environment:

- 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.
- Using a grammar to specify a visual language provides a precise definition of the syntax of the language.
- 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 4 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

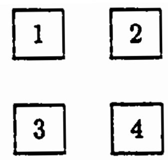

Figure 1: A Two-dimensional Collection of Objects

the two phrases. For a picture, the composition is more complicated than concatenation. The types of compositions may be classified into three 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.
- The embedding of one object into another, such as a text string contained within a box.

Using two-dimensional composition operators, a picture may be decomposed into sub-pictures, down to the primitive elements. The distinctions between the visual languages in Figure 6 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 1 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. For visual languages, the underlying structure is often a graph rather than a tree [14]. A sub-shape may compose with several other shapes in a picture.

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.

grammar where the productions correspond to picture composition operators. The attributes in a PLG represent the spatial information for a picture element (or aggregate). The constraints in a production specify the relationship between the components and the semantic functions compute the information for the aggregate object. Specifically, a picture layout grammar is an attribute multiset grammar where the following conditions hold:

1. Every vocabulary symbol has four attributes L_1, R_1, L_2, R_2 which describe the area occupied by the symbol.

2. Every production is either of the form A $\rightarrow$ {a} (generating a terminal symbol), or A $\rightarrow \{op_i(B, C, ...)\}$ where op_i is a composition operator.

The production operators defined for picture layout grammars are shown in Figure 5. Each production enforces the appropriate conditions on the RHS. For example, the production $A \rightarrow \{B \text{ over } C\}$ shown in Figure 2 has the constraint that $B.L_2 \geq C.R_2$, which ensures that B is located above C.

Each of the productions described above may have additional semantic and constraint functions. To make the analysis tractable, the constraints may only be computed from *totally synthesized* attributes, where totally synthesized attributes are attributes which are either

1. Synthesized attributes of terminal symbols.

2. Synthesized attributes which are computed from totally synthesized attributes.

Finally, to allow picture layout grammars to describe 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 non-tree edge in the parse structure. This allows a sub-shape to be referred to by (i.e composed with) more than one other sub-shape. We make the restriction that the non-remote symbols form a tree, and that all remote references correspond to elements of the tree.

3.1 Example - Nassi-Schneiderman Diagrams

** Example deleted for space reasons - the full paper will contain an example **

4 Spatial Parsing

Our approach to parsing works in several phases as shown in Figure 3. To begin with, we have a set of primitive graphical elements. The first step is to tranform the primitive elements into an attributed multiset of terminal symbols. This mapping is defined by specifying for each primitive class, what terminals should be generated and how the attributes should be synthesized. Normally each primitive element will map to a single terminal symbol.

The *Build* phase analyzes the picture and constructs a *factored multiple derivation (FMD)* structure. The FMD structure represents a set of possible parses, and is based on the factored parse tree model used

Definition 3 *A* factored parse tree *is a set of factored parse tree nodes T such that the following is true:*

1. *If $F \in T$ is a non-terminal node, then for every child list C_j, the reference r_i is a pointer to a complete factored parse tree node $F' \in T$ where the symbol class of F' is the same as the i'th element of the RHS of the production labeling F.*

2. *There is one distinguished node ROOT $\in T$ which is not pointed to by any reference of another node in T.*

3. *Every node in T other than ROOT is pointed to by at least one reference of another node in T.*

4. *No two nodes $F, F' \in T$ have both the same label and the same extent.*

The yield of a factored parse tree is the multiset of terminal symbols found at its leaves. A factored parse tree T is *overlapping* if there are two nodes $F, F' \in T$ such that there are two paths $(S_0 S_1 ... S_m)$ and $(S'_0 S'_1 ... S'_{m'})$ from F to F', where $S_0 = (C_i, r_j)$ and $S'_0 = (C_{i'}, r_{j'})$ and $C_i = C_{i'}$ and $r_j \neq r_{j'}$. That is, there are two paths from F to F' which go through different elements of the same child list of F.

A factored parse tree represents a set of possible derivations. Each child list is an alternative derivation which uses the same production and covers the same extent. If a factored parse tree is nonoverlapping, a parse tree (unfactored) may be extracted from it by starting at the root and choosing any child list, then continuing with all the nodes referenced by that child list and so on until the leaves of the tree are reached. If the yield of the extracted parse tree T' is the same as the yield of the factored parse tree T, then T' is said to *span* T.

Definition 4 *A* factored multiple derivation structure (FMD) *is a set of factored parse tree nodes T such that every node $F \in T$ is the root of a factored parse tree in T.*

4.2 Building the FMD

The parsing algorithm constructs the FMD structure bottom-up by operating directly on the factored parse tree nodes. The algorithm keeps a list of all FPTNs generated, set initially to the attributed multiset of terminal symbols. The nodes are divided into two queues, one of those already processed and one for those not yet processed. Initially all nodes are unprocessed.

As each node is removed from the queue of nodes to be processed, all productions are found which could have this node on the RHS, either alone or in combination with another node from the processed queue. A new FPTN (or a new child list on an existing FPTN) is created for the production. If it is a new FPTN (as opposed to a new child list), it is added to the unprocessed queue. When all applicable productions have been considered, the (original) node is moved from the unprocessed to the processed queue. When all nodes have been moved to the processed queue the algorithm is completed.

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 [19]. The SIL project does demonstrate 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 [20]. Shaw's *Picture Description Language* used a context free grammar augmented by concatenation operators to describe a picture [21]. Bunke has described the use of *attributed programmed graph grammars* for interpreting diagrams.[22] Fu has suggested the use of an attribute grammar for picture recognition [23]. 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 languages, a parsing algorithm and an environment based on a spatial parser. A prototype environment has been developed at Brown University. The prototype environment consists of a component for defining picture layout grammars, a spatial parser and a graphical editor. The environment has been used to define a number of visual languages and parse visual programs. The parser has also been used within a visual programming environment developed at GTE Laboratories [24]. Our continuing research is integrate the semantic processing into the environment.

References

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

[2] Shi-Kua Chang, Tadao Ichikawa, and Panos A. Ligomenides, editors. *Visual Languages*. Plenum Press, 1986.

[3] Nan C. Shu. *Visual Programming*. Van Nostrand Reinhold Company, 1988.

[4] Georg Raeder. *Programming in Pictures*. PhD thesis, University of Southern California, 1984.

[5] Ephraim P. Glinert and Steven L. Tanimoto. Pict: an interactive graphical programming environment. *IEEE Computer*, 17(11):7–25, November 1984.

[6] Robert V. Rubin, Eric J. Golin, and Steven P. Reiss. Thinkpad: A graphical system for programming by demonstration. *IEEE Software*, 2(2):73–79, March 1985.

[7] Robert J. K. Jacob. A state transition diagram language for visual programming. *IEEE Computer*, 18(8):51–59, August 1985.

[8] Takayaki Dan Kimura. Julie W. Choi, and Jane M. Mack. A visual language for keyboardless programming. Technical Report WUCS-86-6, Washington University, March 1986.

A Appendix - additional figures

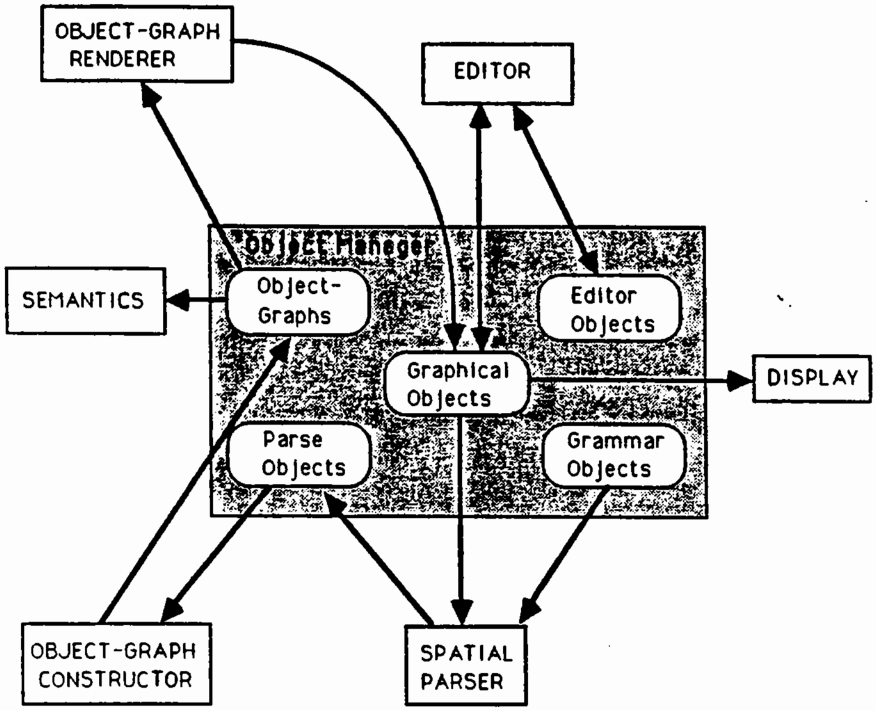

Figure 4: A Visual Programming System based on Spatial Parsing

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 contain an arbitray number of B objects
adjacent_to(B,C)	B is adjacent to (any direction) C
touches_L(B,C)	the (L_1, L_2) point of B is on the boundary of C
touches_R(B,C)	the (R_1, R_2) point of B is on the boundary of C
points_from(B,C)	the L end of B is on C
points_to(B,C)	the R end of B is on C
labels(B,C)	B is adjacent to the line C

Figure 5: Picture Layout Grammar Production Operators