5

Positional Grammars: A Formalism for LR-Like Parsing of Visual Languages

Gennaro Costagliola
Andrea De Lucia
Sergio Orefice
Genny Tortora

ABSTRACT Positional grammars naturally extend context-free grammars for string languages to grammars for visual languages by considering new relations in addition to string concatenation. Thanks to this analogy, most results from LR parsing can be extended to positional grammars while preserving its well known efficiency. The positional grammar model is the underlying formalism of the VLCC (Visual Language Compiler-Compiler) system for the automatic generation of visual programming environments. VLCC inherits and extends to the visual field, concepts and techniques of compiler generation tools like YACC. Due to their nature, the positional grammars are a very suitable formalism for processing languages integrating visual and textual constructs.

5.1 Introduction

Much recent research is focusing on formal methods for the definition and implementation of visual programming environments. In particular, a number of grammar formalisms to describe and parse visual languages have been introduced. In general, the broader the class of languages to be treated is, the less efficient the parsing algorithm is. Due to this, efforts are being made to characterize a class which is expressive enough and, at the same time, efficient to parse. In addition, some efforts are being made in order to classify grammar models according to characteristics such as their expressive power, the way they represent the input, and the parser technology they support.

A first attempt to classify the existing approaches according to their expressive power can be found in [21]. There, several formalisms are compared and classified according to their formal properties in order to provide a comprehensive hierarchy of visual languages.

Basically, two approaches for representing visual language sentences can be distinguished. One of these, the *relation-based representation*, describes a sentence as a set of graphical objects and a set of relations on them [23, 22, 13]. The other, *the attribute-based representation*, conceives a sentence as a set of attributed graphical objects [14, 24, 20, 19, 6].

Many of the proposed formalisms support *order-free pictorial parsers* that process the objects in the input according to no ordering criterion. The formalisms of picture layout grammars [14], relation grammars [11, 23], and constraint multiset grammars [17, 20] fall into this class. In general, and in the worst case, an order-free parser proceeds with a purely bottom-up enumeration. To limit the parsing computational cost, subclasses of PLGs, CMGs, and RGs have been defined to provide the corresponding parsers with predictive capabilities that restrict the search space. To further improve parsing efficiency, *predictive pictorial parsers* such as the pLR parser [9, 6], based on positional grammars, and the one presented in [24], based on relational grammars, have also been defined.

Since their introduction [5], positional grammars have been in continuous evolution, in order to represent broader and broader classes of visual languages. Originally, they were conceived as a simple extension of context-free string grammars to the case of two-dimensional symbolic languages. The generated sentences were basically matrices of symbols and, among the others, could describe simple two-dimensional arithmetical expressions.

The definition of positional grammars has been strongly influenced by the need of having an efficient parser able to process the generated languages. Due to their analogy with string grammars, it has been natural the use of LR parsing techniques. The result was the definition of the *positional LR* (or *pLR*) parsing methodology. This methodology has evolved in parallel with the grammar formalism.

Costagliola et al. [6] introduce a characterization of the visual language syntax based on the nature and interrelations of the objects forming a visual sentence. This characterization has allowed the generalization of the basic concepts of positional grammars to the case of plex languages [12]. Plex sentences are particular types of graphs where each node has a limited number of attaching points used to connect to other nodes. Flowchart diagrams are an example of plex language. According to the syntax characterization of visual languages, each graphical object in a visual sentence presents a set of attributes; a set of relations can then be defined on such attributes. As an example, two-dimensional symbolic sentences are composed by arranging symbols on the screen according to *spatial relations* between their *position*, while plex sentences are obtained by *connecting* graphical objects through their *attaching points*. Here, *position* and *attaching points* are attributes while *spatial relations* and *connections* are relations defined on them, respectively. A complete treatment of these topics can be found in [9].

In [7], to let positional grammars deal with broader classes of graph languages, the formalism has been enhanced with a new derivation mechanism: during the generation of a sentence, tokens are allowed to be shared. On the pLR parsing side, the enhancement consists in allowing the parser to process the same token more than once. This technique, similar to that used in [14], has made positional grammars and pLR parsing able to model and parse visual languages such as transition diagrams of finite state automata.

Positional grammars have, then, followed the attributed-based approach in the representation of the input and the predictive parsing approach for processing visual languages. Being anchored to the LR parsing techniques, this approach is less general than others. However, we feel that positional grammars can be used as basis for the implementation of visual languages in the same way as context-free string grammars are for traditional programming languages. In fact, so far, this approach has been successfully used to define and implement visual languages for two-dimensional arithmetical expressions, structured and semi-structured flow-charts, document layouts, chemical structures, entity relationship diagrams, data-flow graphs, electrical circuits, finite automata, and petri net graphical layouts. Moreover, positional grammars and pLR parsing have the nice feature of naturally defining and parsing languages integrating both visual and textual constructs. In fact, when text is to be processed, they reduce to traditional string grammars and LR parsing, respectively.

The positional grammar model is the underlying formalism of the VLCC (Visual Language Compiler-Compiler) system [10] for the automatic generation of visual programming environments. VLCC uses positional grammars to define a visual language and the pLR based methodology to generate visual language compilers. It adopts concepts and techniques of compiler generation tools like YACC [18] and extends these to the visual field.

Other tools for the automatic generation of visual language compilers from grammar specifications have also been recently proposed. Examples include the Spatial PARser GENerator (SPARGEN) [15], which automatically generates compilers for visual programming languages from an object-oriented picture layout grammar specification, and the user interface generator based on constraint multiset grammars presented in [3], where the visual editor is a general graphical editor with an integrated incremental parser. Unlike VLCC, the design of these tools do not include a graphical editor for the visual definition of grammar specifications, neither they support the integration of visual and textual constructs in the same language.

In this chapter, we illustrate our approach to the visual language syntax, describe the basic positional grammar model, and summarize the pLR parsing methodology as in [9]. Then, we describe the basic characteristics of the VLCC system. Finally, we present an example for the translation of a semi-structured flowchart from its graphical representation into Pascal-like source code.

5.2 Visual Languages

A visual language is conceived as a collection of pictures obtained by arranging graphical objects in two or more dimensions. Visual languages can be syntactically modeled through a vocabulary (the set of graphical objects allowed into a picture of the language), a set of relations used to compose pictures, and a set of rules describing the set of pictures belonging to the language. The graphical objects of a visual language vocabulary are characterized by a set of attributes. Attributes of a graphical object can be distinguished in *graphical attributes*, *syntactic attributes*, and *semantic attributes*.

Graphical attributes characterize the image of the object. Typical graphical attributes are position, size, shape, color, name, etc. Syntactic attributes are used to relate graphical objects in order to form pictures (or visual sentences). A set of graphical objects forms a picture once all the syntactic attributes have been instantiated. Syntactic analysis techniques can be used to check whether the syntactic attributes of the graphical objects in a picture are instantiated according to a proper set of relations, i.e., to check whether a picture belongs to a visual language. Semantic attributes are used to associate semantics with a graphical object; they can be used either to translate a picture into a target format in a syntax-direct fashion or to execute a picture.

The type of syntactic attributes associated with the graphical objects of a visual language and the type of relations that can be used to compose pictures are strongly related and define a *syntactic model*. As an example, let us consider the picture in Figure 5.1 representing a flowchart. It can be modeled as the interconnection of the graphical objects *start*, *predicate*, *function*, and *halt*.

Each object has a set of pre-defined attaching points as syntactic attributes. The attaching points are connected through polylines which visually depict the control flow relation among objects. In this case, the semantics associated with the attaching points of graphical objects implicitly defines the direction of the connections.

An alternative representation for a picture is its *absolute representation* which contains all the objects in the picture together with their syntac-

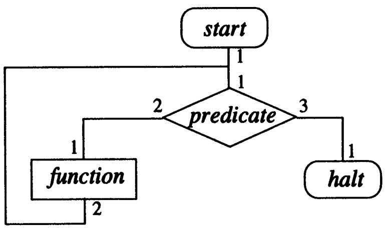


FIGURE 5.1. A Flowchart.

object name	attaching point 1	attaching point 2	attaching point 3
start	*a*	-	-
predicate	*a*	*b*	*c*
function	*b*	*a*	-
halt	*c*	-	-

FIGURE 5.2. An Alternative Representation of the Flowchart in Figure 5.1.

tic attribute values. Figure 5.2 shows the absolute representation of the flowchart in Figure 5.1. In this case, the syntactic attribute values *a*, *b*, and *c* correspond to the three interconnections (*start*, *predicate*, *function*), (*predicate*, *function*), and (*predicate*, *halt*), respectively. For example, the entry (*function*, **attaching point 1**) $= b$ means that the polyline corresponding to *b* is connected to the attaching point 1 of the object *function*.

While the absolute representation of a picture takes into account the values of the syntactic attributes, its *relative representation* considers the relations existing among the graphical objects through their syntactic attributes. In particular, the relative representation of a picture is a string-like notation alternating graphical object names to relation identifiers. As an example, the relative representation of the flowchart above follows:

start $\mathbf{JOINT}^0(1,1)$ *predicate* $\langle\mathbf{JOINT}^0(2,1), \mathbf{JOINT}^0(1,2)\rangle$ *function* $\mathbf{JOINT}^1(3,1)$ *halt*

The number i appearing as superscript of a relation identifier, say **REL**, indicates that **REL** holds between the $(i+1)$th preceding graphical object in the string and the next object. For example, $\mathbf{JOINT}^1(3,1)$ relates the attaching point 3 of the object *predicate* to the attaching point 1 of the object *halt* through the relation **JOINT**. Note that the composite relation $\langle\mathbf{JOINT}^0(2,1), \mathbf{JOINT}^0(1,2)\rangle$ indicates that both joint relations hold between *predicate* and *function*. Even though the relative representation above presents only binary relations, in the flowchart of Figure 5.1 the polyline corresponding to *a* represents a ternary relation among the objects *function*, *predicate* and *start*. This means that the use of binary relations to represent a picture does not limit the degree of the relations in the pictures of a visual language. In fact, an n-ary relation in a picture, with $n > 2$, can always be represented by $n - 1$ binary relations. For instance, the ternary relation corresponding to *a* is represented in the relative representation above by the following two binary relations:

- **JOINT**(1,1) holding between *start* and *predicate*
- **JOINT**(1,2) holding between *predicate* and *function*.

It is worth stressing that it is always possible to provide a relative representation for a picture provided that the two following conditions are satisfied:

- any n-ary relation in the picture, with $n > 2$, can be replaced by a set of semantically equivalent binary relations;
- any binary relation can be replaced by an inverse relation, by preserving the semantics of the picture.

The need for the first condition has already been discussed for the flowchart example. The second condition arises from the need to linearize loops. For example, the relation **JOINT**(1,2) holding between *predicate* and *function* in Figure 5.1 is actually the inverse of the relation **JOINT**(2,1) denoting a flow of control between *function* and *predicate*. In this case the semantics of the attaching points of a graphical object in a flowchart allows to use either of the two relations preserving the control flow semantics.

Different syntactic models can be defined varying on the type of syntactic attributes of graphical objects and on the relations used to compose pictures. For example, the picture in Figure 5.3 shows the layout of the front page of a two-column paper. This sentence belongs to a visual language whose graphical objects are rectangular boxes. In this case, the coordinates of the upper-left and lower-right points of the rectangle can be used as syntactic attributes; they give information about the size and the position of a graphical object. Moreover, geometric relations, such as horizontal and vertical concatenation, and spatial inclusion can be used to compose pictures.

The two examples shown in Figures 5.1 and 5.3, respectively, reflect the two basic modalities that can be used to compose pictures, either connecting or spatially arranging graphical objects. In the first case, the syntactic attributes of graphical objects are instantiated by means of explicit relations (the connections between them). In the second case, the syntactic attributes are automatically instantiated once a graphical object is placed in the Cartesian plane, while implicit relations between two objects can be derived from their relative positions.

It is worth noting that the same visual language can be syntactically modeled in different ways. The choice of the syntactic model used to specify a visual language type depends on its intrinsic characteristics: for example,

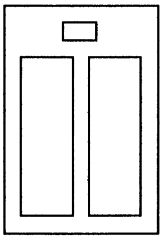


FIGURE 5.3. A Sample Box Sentence.

for graph languages the connection based syntactic model is more natural than the geometric based model. More information about the use of syntactic models for the implementation of visual languages can be found in [8].

5.3 Positional Grammars

Positional grammars are a direct extension of context-free string grammars where more general relations than string concatenation are allowed.[1]

Definition 1 (positional grammar) *A context-free* positional grammar *PG is a six-tuple (N, T, S, P, POS, PE) where:*

N is a finite non-empty set of non-terminals
T is a finite non-empty set of terminals, with $N \cap T = \phi$
$S \in N$ *is the starting non-terminal*
P is a finite set of productions
POS is a finite set of binary relation identifiers
PE is a pictorial evaluator

Both terminals and non-terminals (grammar objects in the following) are graphical objects as defined in Section 5.2.

Each production in P has the following form:

$$\mathrm{A} \rightarrow \mathrm{x}_1\ \mathrm{R}_1\ \mathrm{x}_2\ \mathrm{R}_2 \ldots \mathrm{x}_{m-1}\ \mathrm{R}_{m-1}\ \mathrm{x}_m,\ \Delta$$

where $m \geq 1$, $\mathrm{A} \in \mathrm{N}$, each $\mathrm{x}_i \in \mathrm{N} \cup \mathrm{T}$ denotes a grammar object and each R_j is a sequence of the form $\langle \mathbf{REL}_1{}^{h_1}, \ldots, \mathbf{REL}_i{}^{h_i}, \ldots, \mathbf{REL}_n{}^{h_n} \rangle$ with $n \geq 1$. Each $\mathbf{REL}_i$ is a relation identifier in POS relating the values of the syntactic attributes of x_{j+1} with the ones of x_{j-h_i}, $0 \leq h_i < j$. In the following, we will denote $\mathbf{REL}_i{}^0$ simply as $\mathbf{REL}_i$. The first relation appearing in a sequence R_j is called *driver relation*, the other relations will be referred to as *tester relations*. Driver relations are used to steer parsing by retrieving the next element from the input, while tester relations are used to check whether the last grammar object parsed (terminal or non-terminal) is correctly related to previously parsed elements in the picture.

Δ is a rule which synthesizes the syntactic attribute values of A from those of $\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_m$. PE is a function which transforms a sentential form derived from the grammar into the corresponding pictorial form.

The following definitions are all meant to be referred to a context-free positional grammar ("positional grammar" for short) PG = (N, T, S, P,

[1] Please note that, in the rest of the chapter, the term *context-free* should be considered as referring only to the form of the grammar rules. A discussion on the meaning of context-freeness for visual languages is out of the scope of this work.

POS, PE).

We write $\alpha \Rightarrow \beta$ and say that β is derived from α in one step, if there exist positional sentential forms δ, γ, η and a non-terminal A such that $\alpha = \gamma A \delta$, $A \rightarrow \eta, \Delta$ is a production in P and $\beta = \gamma[\eta\Delta]\delta$. The square brackets maintain the precedence in the application of the pictorial evaluator PE. We write $\alpha \Rightarrow^* \beta$ and say that β is derived from α, if there exist strings $\alpha_0, \alpha_1, \ldots, \alpha_m$ $(m \geq 0)$ such that $\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_m = \beta$.

The sequence $\alpha_0, \alpha_1, \ldots, \alpha_m$ is called a derivation of β from α. Let x be a grammar object:

- a *positional sentential form* from x is a string β such that $x \Rightarrow^* \beta$.
- a *positional sentence* from x is a positional sentential form from x which does not contain non-terminals.
- a *pictorial form* (*picture*, respectively) from x is the result of the evaluation of a positional sentential form (positional sentence, respectively) from x by PE.

As usual, L(PG), the language generated by PG, denotes the set of the *pictures* from the starting symbol S.

Intuitively, a positional sentence is a relative representation of a picture in L(PG), and the pictorial evaluator PE produces a picture from its relative representation.

As an example, let us consider the following positional grammar which generates tree-structured data-flow graphs. The terminals and the productions of the grammar are graphically depicted in Figure 5.4 and 5.5, respectively.

N = {DFL, IF }
T = {op, arg, test, mux}
S = DFL
POS = {**JOINT**}
P = { 1) DFL $\rightarrow$ arg $\quad (1) \equiv (1)$
2) DFL $\rightarrow$ op 1_1 DFL 2^1_1 DFL′ $\quad (1) \equiv (300)$
3) DFL $\rightarrow$ mux 1_1 DFL 2^1_1 DFL′ 3^2_1 IF $\quad (1) \equiv (4000)$
4) IF $\rightarrow$ test 1_1 DFL 2^1_1 DFL′ $\quad (1) \equiv (300)$
}

The object op represents the operation node, where the string "OP" labelling the object is an operator from the set {'+', '*', '-', '/'}. The condition node is represented by the object test, where "TEST" is in {'<', '>', '≤', '≥', '=', '≠'}. Attaching points 1 and 2 of the objects op and test denote the inputs of the nodes, while attaching point 3 denotes the output. The object arg is the argument node for the constants and variables of a data-flow graph (for example, "ARG" may indicate a constant or variable name like 1, 2 or x, y), while mux describes the multiplexer node. Here,

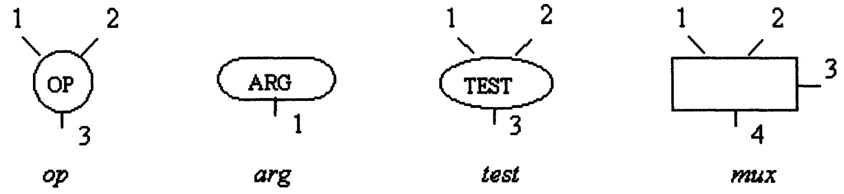


FIGURE 5.4. The Terminals of the Data-Flow Graph Grammar.

attaching points 1, 2, 3, and 4 identify the input lines true and false, the select line, and the output line, respectively.

POS contains only one identifier, **JOINT**, for a relation of the type **JOINT**(h, k) defined as follows: given two grammar objects denoted by x and y, the relation x **JOINT**(h, k) y holds iff the attaching point h of x is connected to the attaching point k of y. In the positional grammar above, a relation $\mathbf{JOINT}^i$(h, k) is written as h^i_k if $i \neq 0$ and as h_k if $i = 0$. The type of syntactic attributes used to model the grammar objects of this language (a fixed number of attaching points) and the relation **JOINT** define a particular type of *connection based syntactic model* [8]; we call such model *plex model* [12].

The productions of the grammar define a data-flow graph DFL either as a simple argument arg, or as an operation op taking inputs from the outputs of two data-flow graphs, or as a multiplexer mux taking as inputs the outputs of two data-flow graphs and of a condition IF. On the other hand, a condition IF is defined as a test taking as inputs the outputs of two data-flow graphs.

For each production $A \rightarrow x_1\ R_1\ x_2 \ldots R_{m-1}\ x_m,\ \Delta$ where A has n attaching points $(1,\ldots,n)$, the synthesizing rule Δ has the form

$$(1, \ldots, i, \ldots, n) \equiv (q_{1_1} \cdots q_{1_m}, \ldots, q_{i_1} \cdots q_{i_m}, \ldots, q_{n_1} \cdots q_{n_m})$$

The value of the ith attaching point of the non-terminal A is named *tie-point*, [12], and is defined by a string $q_{i_1} \ldots q_{i_m}$ of attaching point identifiers; each q_{i_j} is either 0 or an attaching point identifier of x_i, $1 \leq i \leq n$, $1 \leq j \leq m$. The value 0 is a place marker not associated to any attaching point. For example, in the second production of the positional grammar above the tie-point 300 defines the attaching point 1 of DFL to be the attaching point 3 of op.

Figure 5.6 shows a picture generated by the positional grammar above and its meaning, respectively.

In order to explain how PE works, let us consider the following derivation:

DFL $\Rightarrow$
$\Rightarrow$ [mux 1_1 DFL 2^1_1 DFL 3^2_1 IF, Δ_3] $\Rightarrow$
$\Rightarrow$ [mux 1_1 [op 1_1 DFL 2^1_1 DFL,Δ_2] 2^1_1 DFL 3^2_1 IF, Δ_3] $\Rightarrow$
$\Rightarrow \ldots \Rightarrow$

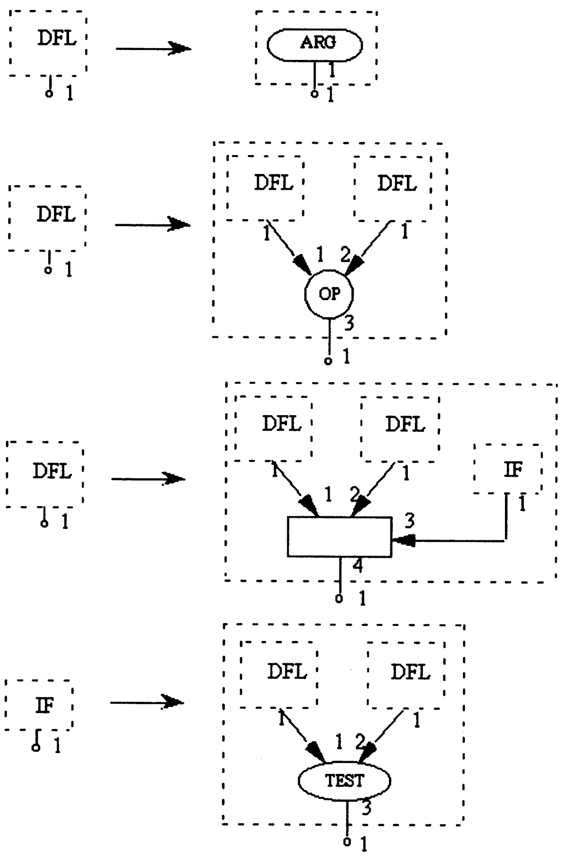


FIGURE 5.5. The Visual Productions for the Data-Flow Graph Grammar.

$\Rightarrow$ [mux 1_1 [op 1_1 [arg, Δ_1] 2^1_1 [arg, Δ_1], Δ_2] 2^1_1 [op 1_1 [arg, Δ_1] 2^1_1 [arg, Δ_1], Δ_2] 3^2_1 [test 1_1 [arg, Δ_1] 2^1_1 [arg, Δ_1], Δ_4], Δ_3]

In the derivation, brackets are used to maintain the derivation order, which is necessary in the application of the pictorial evaluator. At each nesting level, PE evaluates the relations **JOINT**, if any, and applies the corresponding synthesizing rule Δ. Whenever X **JOINT**(h, k) Y must be evaluated, PE spatially arranges X and Y such that the attaching point h of X is connected to the attaching point k of Y. Note that X and Y can be grammar objects or partial results of PE.

By applying PE to the sentential form

[mux 1_1 [op 1_1 DFL 2^1_1 DFL, Δ_2] 2^1_1 DFL 3^2_1 IF, Δ_3]

the pictorial form in Figure 5.7 is obtained.

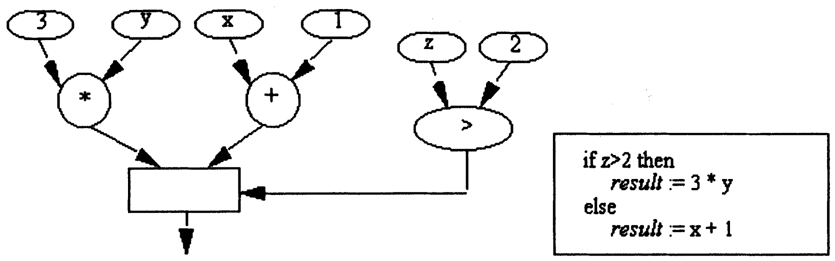


FIGURE 5.6. A Data-Flow Graph and Its Meaning.

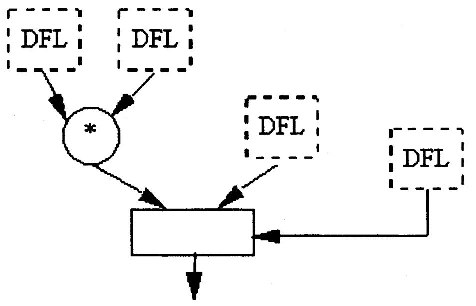


FIGURE 5.7. A Data-Flow Graph.

It is worth noting that the synthesizing rule Δ and the pictorial evaluator PE depend on the particular syntactic model. For example, in the case of the *plex model*, PE is a function that takes as input a positional sentential form Σ from a grammar object x and yields as output a pictorial form Π obtained as follows:

```
if $\Sigma$ = x is a grammar object then $\Pi$ is the grammar object x itself
 else
 begin
  let $\Sigma = \Sigma_1\ R_1\ \Sigma_2\ R_2 \ldots R_{m-1}\ \Sigma_m, \Delta$
   where each $\Sigma_i$ is a positional sentential form from a grammar
   object $x_i$
  if $\Pi_i$ is the pictorial form from $x_i$ (i=1...m, respectively),
   obtained by applying PE on $\Sigma_i$ (i=1...m, respectively)
    then $\Pi$ is obtained by interconnecting $\Pi_1, \Pi_2, \ldots, \Pi_m$
     according to the following rule:
     for each $\mathbf{JOINT}^i$(h, k) occurring in $R_j$ with $0 \leq i \leq j-1$
     and $1 \leq j \leq m$, connect attaching point h of $\Pi_{j-i}$ (i.e., $x_{j-i}$)
     to attaching point k of $\Pi_{j+1}$ (i.e., $x_{j+1}$)
  The attaching points of $\Pi$ are calculated according to $\Delta$
 end
```

5.4 Positional LR Parsing of Visual Languages

The positional grammar model supports the construction of an efficient parser by means of the well known LR parsing technique [1]. In this section we briefly illustrate the basic concepts of positional LR parsing. Details can be found in [9]. Figure 5.8 shows the general schema of a positional LR parser.

In the pLR parsing methodology, the input access is no longer sequential but driven by the relations contained in the positional grammar. This is implemented by adding a new column, named *next*, to the LR parsing table. The column *next* associates a pair (**REL**^{i}, x) to each parser state I_k, where **REL** is a relation and x is a grammar object. Whenever the parser reaches state I_k, the pair (**REL**^{i}, x) is used to select from the input the next object to be processed by the parser. This technique is called *syntax-directed scanning* of the input, first defined in [4] for two-dimensional symbolic languages.

The input to the parser is a dictionary storing the absolute representation of a picture. No parsing order is defined on the graphical objects in the dictionary. The parser retrieves the objects in the dictionary by *find* operations driven by the relations in the column *next* of the parsing table. In this way, the parser implicitly builds a relative representation from the input absolute representation. Analogously to the usual end-of-string marker, the end-of-input symbol $ is returned to the parser if and only if the input has been completely visited, i.e., each token has been parsed and all the explicit relations have been walked.

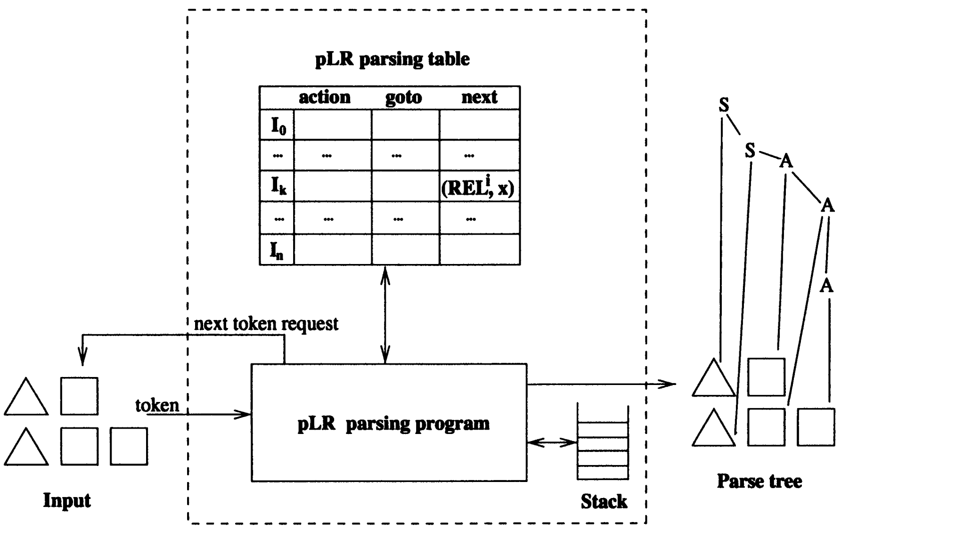


FIGURE 5.8. The pLR Parsing Model.

The pLR parser uses a stack-like data structure which behaves as a stack except that elements below the top may also be read. Each entry in the stack will contain either a grammar object or a state. Then, the content of the stack is of the form $s_0 X_1 s_1 X_2 \ldots X_m s_m$, where each X_i is a grammar object together with its syntactic attributes, each s_i is a state and s_m is on the top of the stack.

A positional LR (pLR) parsing table consists of three parts: the *action* and *goto* parts of the LR parsing tables and the *next* part (see Figure 5.9). The column *next* is used to implement the syntax directed scanning of the input. Whenever the parser reaches state I_k, the pair ($\mathbf{REL}_i$, x) in the entry *next*$[I_k]$ is used to select from the input the next object to be processed by the parser. The *action* and *goto* parts of a pLR parsing table are similar to those of an LR(0) parsing table for string languages [1]. The reduce actions do not depend on the next grammar object to be parsed but only on the current state, i.e., if a reduction is required in state I_k, then all the entries in the corresponding row of the *action* part will contain such a reduction. Moreover, the corresponding entry in the column *next* is not defined, since this entry is used only for shift actions.

Shift and goto actions in a pLR parsing table are of the form "R: shift state" and "R: state," respectively, where R is a sequence, possibly empty, of relations and it is referred to as an *action condition* in the following. A shift or goto action is performed only if the corresponding action condition evaluates to true or it is empty. In general, if a sequence $\langle \mathbf{REL}_1{}^{h_1}, \ldots, \mathbf{REL}_i{}^{h_i}, \ldots, \mathbf{REL}_n{}^{h_n} \rangle$ appears in the right-hand side of a production of a positional grammar, the driver relation $\mathbf{REL}_1{}^{h_1}$ will be stored in the column *next* of the parsing table and used in the syntax-directed scanning of the input, while the sequence of tester relations $\langle \mathbf{REL}_i{}^{h_i}, \ldots, \mathbf{REL}\ _n{}^{h_n} \rangle$ will be used as action condition in the *goto* or *action* part of the table.

Figure 5.9 shows the pLR parsing table for the data-flow graph positional grammar described in the previous section. The notations r*i* and s*j* indicate the actions "reduce using production rule *i*" and "shift token and go to state *j*," respectively. For example, if the parser is in state 1 and the next symbol is \$, then the parser accepts the input (action *acc*), while if the current state is 2, then the parser executes action r1, whatever the next input symbol is. As usual, an empty entry indicates a syntax error. Due to the simplicity of the grammar, all action conditions in the table are empty: for instance, if the current state is 3 and the next symbol is "arg," then the parser directly shifts the symbol onto the stack and goes to state 2, while if the state resulting from the reduction of a DFL non-terminal is state 3, then the parser directly goes to state 5.

The entry SP in the column *next* is a special identifier and it is used at the beginning of the parsing process to retrieve the first object to be parsed. The entry ANY is another special identifier and is used to check whether the whole input has been parsed. Only in this case the end-of-input marker \$ should be processed by the parser. In the parsing table in Figure 5.9, the

state	action					goto		next
	arg	op	test	mux	$	DFL	IF	
0	:s2	:s3		:s4		:1		**SP**
1					acc			**ANY**
2	r1	r1	r1	r1	r1			
3	:s2	:s3		:s4		:5		(1_1, DFL)
4	:s2	:s3		:s4		:6		(1_1, DFL)
5	:s2	:s3		:s4		:7		(2^1_1, DF L)
6	:s2	:s3		:s4		:8		(2^1_1, DF L)
7	r2	r2	r2	r2	r2			
8			:s10				:9	(3^2_1, IF)
9	r3	r3	r3	r3	r3			
10	:s2	:s3		:s4		:11		(1_1, DFL)
11	:s2	:s3		:s4		:12		(2^1_1, DF L)
12	r4	r4	r4	r4	r4			

FIGURE 5.9. An Example of the pLR Parsing Table.

identifier SP refers to the node in the data-flow graph producing the final output. For example, in the data-flow graph of Figure 5.6 this node is the node mux.

The time complexity of the parsing algorithm depends on the number of objects in the input, on the particular syntactic model and on the grammar. In [9], it has been shown that for a fixed grammar the time complexity is $O(n(t_{sm}))$, where t_{sm} depends on the implementation of the particular syntactic model, in particular on the implementation of the input access operation *find* and of the synthesizing rule Δ. For example, for visual languages in the *plex* syntactic model, the time complexity for synthesizing an attaching point is constant, while the time of the access operation *find* may vary from a constant to $O(n)$, depending on the chosen implementation of the input dictionary. If proper hashing techniques are used in the implementation of the input dictionary, the expected time complexity reduces to $O(n)$.

5.5 The VLCC System

In this section we show how a visual language can be implemented according to its syntactic model by using the Visual Language Compiler-Compiler (VLCC) system [10].

VLCC is a tool for the automatic generation of visual programming environments implementing visual languages. A prototype of the VLCC system has been implemented using object-oriented technology under MSWindow 3.x^{TM} and MSWindows95/NT^{TM}. Given a complete language specification, VLCC generates an integrated environment consisting of a graphical editor for the definition of sentences and a compiler for their analysis and translation. Each new VLCC generated environment is structured in three levels:

- the graphical interface
- the syntactic model implementation
- the language compiler and tokens

Each level is implemented by a module consisting of a set of C++ classes that inherit or use classes from the higher levels.

The first level is common to all the visual language environments generated by VLCC: it mostly contains abstract classes implementing the general framework of the graphical editor. For example, this module includes virtual functions defining the way the graphical editor draws a token and functions implementing command menus such as copy, paste, undo, etc.

The second level implements a syntactic model; it is common to all the VLCC generated environments for visual languages which have been modeled according to that syntactic model. This module contains classes implementing the picture representation and the relations of interest. As shown above, the picture representation is given by the syntactic attributes of the tokens of the picture. The syntactic models so far implemented allow the implementation of graph languages such as finite automata, plex languages like flowcharts, and simple box languages. However, thanks to the object-oriented architecture of the system, implementations of other syntactic models can be easily added.

The third level is specific to each language and contains C++ classes implementing the compiler and the tokens for that language. The compiler is built on the syntax and semantics specification given through a context-free positional grammar augmented with semantic rules. It is automatically generated in an LR-like fashion according to the pLR parsing methodology. A C++ class implementing a token contains information regarding both its graphical aspect and the association between its syntactic attributes and its image. Moreover, it inherits all the functions for graphical token manipulation from the above levels.

While the first two levels are pre-defined and implemented in dynamic libraries, the third level is built by a visual language designer/implementor through the use of the VLCC interface. A VLCC user selects a syntactic model, defines a positional grammar through a production editor and creates the graphical objects through a symbol editor. He/She then runs the VLCC code generator to automatically generate the C++ classes corresponding to the third level of the VLCC generated environment architecture. Finally, the executable code for the visual programming environment is obtained by compiling the output C++ code with the pre-defined libraries.

The generated graphical environment allows a user to draw and compile pictures through two palettes: one to set the language graphical objects on the screen and one to instantiate their syntactic attributes. The latter palette includes also buttons to annotate graphical objects with textual sentences.

5.6 An Example

In this section, we show an example of application of positional grammars for the translation of a semi-structured flowchart from its graphical representation into Pascal-like source code. This visual language has been developed starting from an example in [16]. Figures 5.10 and 5.11 show the terminals and their attaching points, and the visual grammar productions, respectively.

The terminals DT and CT represent a Boolean condition and an instruction, respectively. The terminal BSTAT indicates the beginning of a compound statement while the terminals JUMPD, JUMPS and UNION have been introduced for managing the loops.

The positional grammar for semi-structured flowcharts follows; it can be derived directly from the visual grammar in Figure 5.11:

N = {fchart, cn, dn}
T = {START, HALT, CT, DT, BSTAT, JUMPD, JUMPS, UNION}
POS = {**JOINT** }
P = { (1) fchart → START 1_1 cn 2_1 HALT
(2) cn → BSTAT 2_1 dn ⟨2_1,3_2⟩ UNION
(cnl, cn2) ≡ (BSTAT1, UNION3)
(3) cn → BSTAT 2_1 JUMPD ⟨2_1,3_3⟩ dn
(cnl, cn2) ≡ (BSTAT1, dn2)
(4) cn → BSTAT 2_1 JUMPS ⟨2_1,3_2⟩ dn
(cnl, cn2) ≡ (BSTAT1, dn3)
(5) cn → cn′ 2_1 cn″
(cnl, cn2) ≡ (BSTAT1, cn″2)
(6) cn → CT
(cnl, cn2) ≡ (CT1, CT2)
(7) dn → cn 2_1 dn′
(dnl, dn2, dn3) ≡ (cnl, dn′2, dn′3)

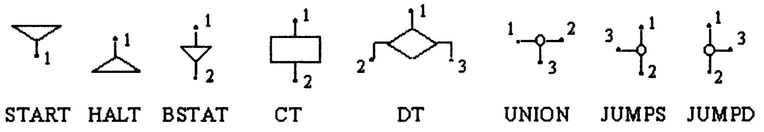


FIGURE 5.10. The Visual Terminals for Semi-Structured Flowcharts.

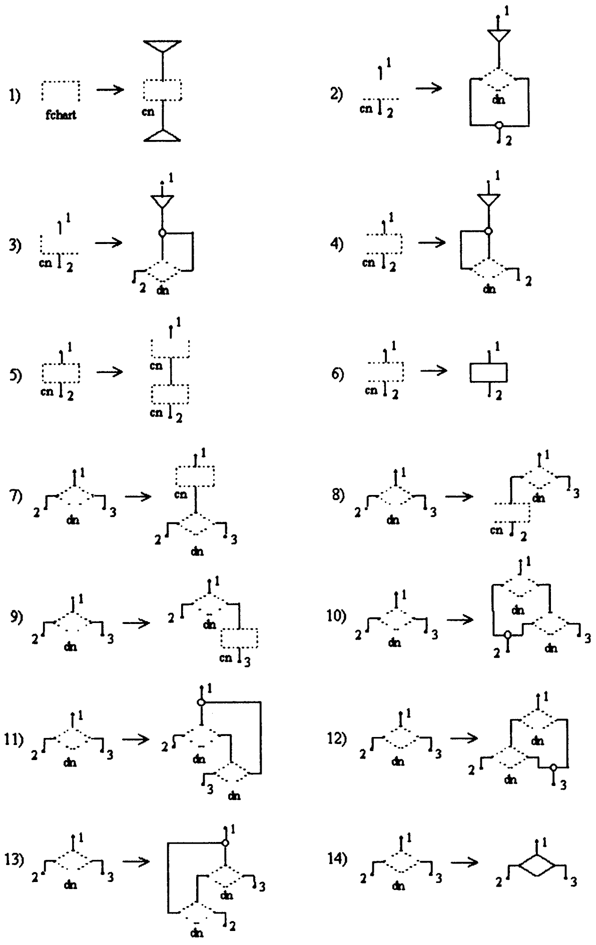


FIGURE 5.11. The Visual Grammar for Semi-Structured Flowcharts.

(8) dn → dn′ ⟨2_1,2_1⟩ cn
(dn1, dn2, dn3) ≡ (dn′1, cn2, dn′3)

(9) dn → dn′ ⟨3_1,3_1⟩ cn
(dn1, dn2, dn3) ≡ (dn′1, dn′2, cn2)

(10) dn → dn′ 3_1 dn″ ⟨2_2,2^1_1,2^1_1⟩ UNION
(dn1, dn2, dn3) ≡ (dn′1, UNION3, dn″3)

(11) dn → JUMPD 2_1 dn′ ⟨3_1,3^1_3⟩ dn″
(dn1, dn2, dn3) ≡ (JUMPD1, dn′2, dn″2)

(12) dn → dn′ 2_1 dn″ ⟨3_1,3^1_2,3^1_2⟩ UNION
(dn1, dn2, dn3) ≡ (dn′1, dn″2, UNION3)

(13) dn → JUMPS 2_1 dn′ ⟨2_1,3^1_2⟩ dn″
(dn1, dn2, dn3) ≡ (JUMPS1, dn′3, dn″3)

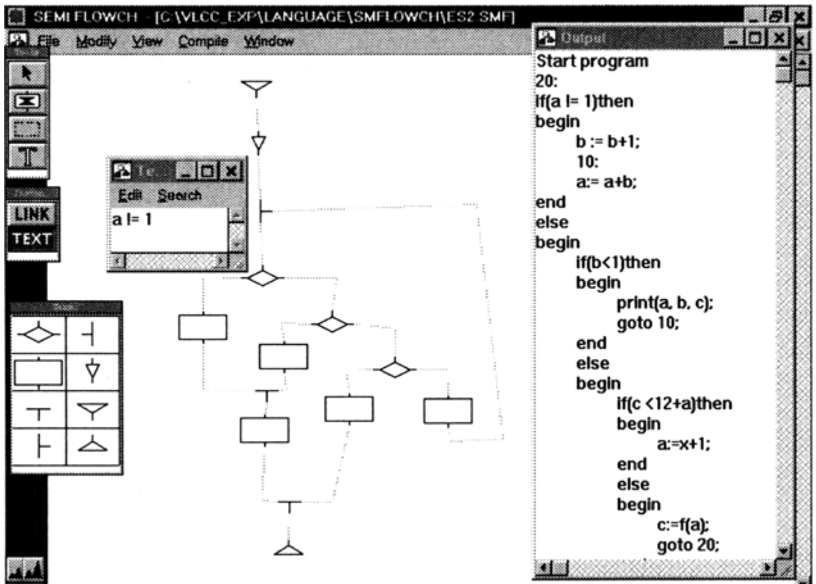
(14) dn → DT (dn1, dn2, dn3) ≡ (DT1, DT2, DT3)

Here the Δ rules have been given in a simplified way. As an example, the Δ rule (dn1, dn2, dn3) ≡ (JUMPS1, dn′3, dn″3) taken from production (13) indicates that the attaching points 1, 2, and 3 of the object dn must be set to attaching points 1 of JUMPS, 3 of dn′ and 3 of dn″, respectively. For sake of simplicity we omit the productions for the text part of the language and the translation rules.

Once these specifications are given to VLCC through its editors, it is possible to automatically produce the visual environment shown in Figure 5.12. The positional grammar is then input to VLCC in its textual notation. In this case, the environment has been used to draw a semi-structured flowchart, and translate it into Pascal-like code.

In such an environment the graphical symbols DT and CT can be assigned a string through the button TEXT. The translation process parses these strings while parsing the picture and uses them to produce the final Pascal-like source code. In Figure 5.12 the highest terminal DT is associated with the string "(a !=1)" as shown in the text editor window on its left.

Other examples of VLCC generated can be found at the VLCC web site http://www.unisa.it/gencos.dir/vlcc.htm.



```
a != 1
```

```
Start program
20:
If(a != 1)then
begin
      b := b+1;
      10:
      a:= a+b;
end
else
begin
      if(b<1)then
      begin
            print(a, b, c);
            goto 10;
      end
      else
      begin
            if(c <12+a)then
            begin
                  a:=x+1;
            end
            else
            begin
                  c:=f(a);
                  goto 20;
```

FIGURE 5.12. The VLCC-Generated Visual Environment for Semi-Structured Flowcharts.

5.7 Conclusions

In this chapter we presented an overview of the positional grammars, the pLR parsing methodology and their use in the VLCC system for producing visual language environments.

As further research, we are focusing on the definition and implementation of syntactic models for the description of dynamic visual languages [2], where graphical objects attributes for handling temporal information need to be introduced.

Since the pLR parsers so far implemented are not incremental, the graphical editor of a VLCC generated environment is not syntax-driven. This makes the system suitable to work with pictures which have been input to the environment through a scanner or through a fast and expert user who does not need any help in formulating the pictures. However, we are defining a new incremental pLR parser based on the extension of traditional LR incremental parsers in order to make the VLCC environments suitable for processing tasks that require interaction or incremental interpretation of pictures.

5.8 References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, New York, 1985.

[2] S.K. Chang. Dynamic visual languages. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, pages 308–315, Boulder, USA, 1996. IEEE Comp. Soc. Press.

[3] S.S. Chok and K. Marriot. Automatic construction of user interfaces from constraint multiset grammars. In *Proceedings of the 1995 IEEE Symposium on Visual Languages*, Darmstadt, Germany, 1995. IEEE Comp. Soc. Press.

[4] G. Costagliola and S.-K. Chang. Parsing linear pictorial languages by syntax-directed scanning. *Languages of Design*, 2(3):229–248, 1994.

[5] G. Costagliola and S.K. Chang. Dr parsers: A generalization of lr parsers. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, Skokie, USA, 1990. IEEE Comp. Soc. Press.

[6] G. Costagliola, A. De Lucia, and S. Orefice. Towards efficient parsing of diagrammatic languages. In *Proceedings of the 1994 International Workshop on Advanced Visual Interfaces*, pages 162–171, Bari, Italy, 1994. ACM Press.

[7] G. Costagliola, A. De Lucia, S. Orefice, and G. Tortora. Efficient parsing of data-flow graphs. In *Proceedings of 7th International Conference*

on Software Engineering and Knowledge Engineering, pages 226–233, 1995.

[8] G. Costagliola, A. De Lucia, S. Orefice, and G. Tortora. A framework of syntactic models for the implementation of visual languages. In *Proceedings of the 1997 IEEE Symposium on Visual Languages*, pages 58–65, Capri, Italy, 1997. IEEE Comp. Soc. Press.

[9] G. Costagliola, G. Tortora, S. Orefice, and A. De Lucia. A parsing methodology for the implementation of visual systems. *IEEE Transaction on Software Engineering, to appear.*

[10] G. Costagliola, G. Tortora, S. Orefice, and A. De Lucia. Automatic generation of visual programming environments. *IEEE Computer*, 28(3):56–66, 1995.

[11] C. Crimi, A. Guercio, G. Nota, G. Pacini, G. Tortora, and M. Tucci. Relation grammars and their application to multi-dimensional languages. *Journal of Visual Languages and Computing*, 2:333–346, 1991.

[12] J. Feder. Plex languages. *Information Science*, 3:225–241, 1971.

[13] F. Ferrucci, G. Pacini, G. Satta, G. Tortora M. Sessa, M. Tucci, and G. Vitiello. Symbol-relation grammars: A formalism for graphical languages. *Information and Computation*, 131:1–46, November 1996.

[14] E.J. Golin. Parsing visual languages with picture layout grammars. *Journal of Visual Languages and Computing*, 2:1–23, 1991.

[15] E.J. Golin and T. Magliery. A compiler generator for visual languages. In *Proceedings of the 1993 IEEE Workshop on Visual Languages*, pages 314–321, Bergen, Norway, 1993. IEEE Comp. Soc. Press.

[16] A. Habel. *Hyperedge Replacement: Grammars and Languages.* PhD thesis, Computer Science Departments of the Technical University of Berlin and the University of Bremen, 1989.

[17] R. Helm, K. Marriott, and M. Odersky. Building visual language parsers. In S.P. Robertson, G.M. Olson, and G.S. Olson, editors, *Human Factors in Computing Systems: CHI'91 Conference Proceedings*, pages 105–112, New York, 1991. Addison-Wesley.

[18] S.C. Johnson. Yacc: Yet another compiler compiler. Technical report, Bell Laboratories, Murray Hills, NJ, 1974.

[19] R. Lutz. Chart parsing of flowgraphs. In *Proceedings of 11th International Joint Conference on Artificial Intelligence*, pages 116–121, 1989.

[20] K. Marriot. Constraint multiset grammars. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pages 118–125, Saint Louis, USA, 1994. IEEE Comp. Soc. Press.

[21] K. Marriott and B. Meyer. The CCMG visual language hierarchy. In *this volume.*

[22] J. Rekers and A. Schurr. A graph grammar approach to graphical parsing. In *Proceedings of the 1995 IEEE Symposium on Visual Languages*, pages 195–202, Darmstadt, Germany, 1995. IEEE Comp. Soc. Press.

[23] M. Tucci, G. Vitiello, and G. Costagliola. Parsing non-linear languages. *IEEE Transactions on Software Engineering*, 20(9):720–739, 1994.

[24] K. Wittenburg. Earley-style parsing for relational grammars. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 192–199, Seattle, USA, 1992. IEEE Comp. Soc. Press.