

Extended Positional Grammars

GENNARO COSTAGLIOLA and GIUSEPPE POLESE

Dipartimento di Matematica ed Informatica
Università di Salerno
84081 Baronissi (SA), Italy
{gencos, giupol}@dia.unisa.it

Abstract

Positional grammars are a formalism for the definition and implementation of visual languages. They have already been used in the past as part of the VLCC system (Visual Language Compiler-Compiler) for the definition and the implementation of visual environments for editing and compiling flowcharts, chemical structures, combinatorial networks, electric circuits, etc. In this paper we introduce the eXtended Positional Grammars (XPG, for short) that enhance the descriptive power of Positional Grammars. We also present a more powerful LR-based methodology for parsing visual languages described by XPGs. The result is the possibility of describing and compiling a much wider class of visual languages yet keeping most of LR parsing efficiency.

Index terms — Visual programming environments, multidimensional languages, LR parsing, positional grammars.

1. Introduction

In the past years, many efforts have been made toward the definition of grammar formalisms to describe and parse visual languages. In general, these formalisms use two different approaches when representing a visual language sentence: the *relation-based* approach, and the *attribute-based* approach. The former describes a sentence as a set of graphical objects and a set of relations on them [7, 11, 12], the latter conceives a sentence as a set of attributed graphical objects [2, 8, 10, 13].

Many of the proposed grammar formalisms support *order-free pictorial parsers* that process the input objects according to no ordering criterion. The formalisms of Picture Layout Grammars [8], Relation Grammars, [12], and Constraint Multiset Grammars, [10], 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*, like the pLR parser [2], based on Positional Grammars, and the one presented in [13], based on Relational Grammars, have also been

defined. An exhaustive survey on visual language current formalisms can also be found in [9].

In general, the broader the class of languages to be treated, the less efficient the parsing algorithm is. Due to this, big efforts are being made to characterize a class that is expressive enough and, at the same time, efficient to parse. This paper abides by this research direction and presents a new more powerful version of positional grammars, namely eXtended Positional Grammars (XPG, for short) and the XpLR parsing methodology, based on XPGs. This methodology is also able to solve a number of conflicts usually arising with former pLR parsers, [2]. As a consequence, XPGs and XpLR parsing describe and parse a class of visual languages wider than the one handled by previous positional formalisms. In fact, the use of this methodology within the latest version of the Visual Language Compiler-Compiler [4] allows the generation of many practical visual environments such as environments used in software engineering and in workflow management.

In Section 2 we give a framework for describing multidimensional languages and restrict it to the case of visual languages. We formalize the concepts of attribute-based, relation-based and linear representation of a visual sentence. In Section 3 we introduce the eXtended Positional Grammars (XPG, for short) and provide a grammar for state transition diagrams as an example. In Section 4, we give a new LR-based algorithm for the parsing of visual languages based on XPGs. This new algorithm, named XpLR(0) parser, is also able to solve a number of conflicts usually arising in pLR parsing tables. Section 5 contains the Conclusions.



2. A framework for describing visual languages

In this section we provide a framework for the description of multidimensional languages and apply it to the context of visual languages as a particular case. To do this we introduce the concepts of generalized symbols and generalized sentences, and give three different ways of representing a sentence.

2.1 Generalized symbols

A generalized symbol gs (gsymbol for short) is defined as a triple (M, S, L) where M is the physical component of the symbol and is needed to materialize gs to our senses, S is the syntactic component used to relate gs to other symbols. This component depends on the context where the symbol is considered. L is the semantic interpretation of gs and is used to derive its meaning or that of a sentence containing it. The three components are not necessarily disjoint.

In particular, M is a set of attributes that specify the physical appearance of the gsymbol such as size, color, shape, etc.; S is a set of attributes, named *syntactic attributes*, whose values depend on the "position" of the gsymbol in a sentence; L is a conceptual structure defining the semantics of the gsymbol. When we say that the three components are not necessarily disjoint we mean that certain attributes may be part of more than one component of a generalized symbol. In general, most syntactic attributes are also part of M .

As an example, let us consider the generalized symbol  of a flowchart. In this case, M describes its graphical aspect, i.e., a rhombus and three little circles; S keeps track of the links connected to each attaching point (the circles). In order not to cause syntactic errors, 1) the gsymbol must be connected to other gsymbols through links leaving the attaching points and 2) no two attaching points in it can be connected. The semantic component L qualifies  as the condition block of a conditional or loop statement in a flowchart.

2.2 Generalized sentences

A generalized alphabet Σ is a set of generalized symbols. A generalized sentence gs on Σ is a set of generalized symbol instances $gs = \{x_1, x_2, \dots, x_n\}$ whose physical and syntactic components are fully instantiated.

As an example, let us consider the generalized sentence "flowchart" in Figure 2.1(i). Here an attaching point of a block is connected to an attaching point of another block.

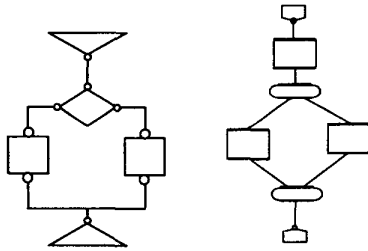


Figure 2.1 A flowchart (i) and an activity diagram (ii)

The information needed to define the context of a generalized symbol may be defined according to the *syntactic model* [3] under consideration. In the following,

we recall the definitions for the String, Plex, Graph and Box syntactic models.

The syntactic model String requires only the attribute *position* to define the context of a symbol, i.e., its position in a sentence. In this model, sentences can be built by combining gsymbols through the only *string concatenation* relation.

The syntactic model Plex [6] defines the context of a generalized symbol through a *sequence of a finite number of attaching points*. In this model, generalized sentences can be built by combining gsymbols through the *connect* relation, i.e., by connecting attaching points of gsymbols through links. An attaching point can only be attached to one link end. The flowchart in Figure 2.1(i) can be easily characterized according to this syntactic model. The circles denote the attaching points.

The syntactic model Graph is an extension of the syntactic model Plex and can be used to model visual languages such as transition diagrams, entity-relationship diagrams, Petri nets, Object Oriented models, and others. Under this model a generalized symbol can be given a *sequence of a finite number of attaching regions* as syntactic attributes. In this model, sentences are built as in the model Plex with the difference that more than one link can be attached to an attaching region. Figure 2.1 (ii) depicts an activity diagram layout as used in UML specifications. It can be characterized by the model Graph where the bold lines denote the attaching regions.

The syntactic model Box defines the context of a generalized symbol through *the coordinates of the upper-left and the lower-right vertices* of its bounding box. In this model, generalized sentences can be built by combining gsymbols through relations such as *containment, overlapping, horizontal and vertical concatenations, etc.* Other syntactic models, denoted as *Hybrid* syntactical models, can be derived by combining the characteristics of the previous models [4].

2.3 Representation of generalized sentences

A generalized sentence can be represented *externally* by materializing all of its gsymbols or *internally* by taking into account the M , S and L components of its gsymbols. Figure 2.2 classifies all the possible representations.

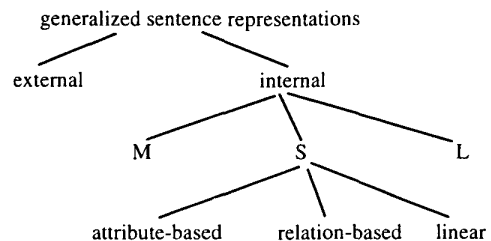


Figure 2.2 A classification of generalized sentence representations

Usually, the M component of each gsymbol is represented as a set of attributes describing the physical characteristics of the gsymbol. In the case of a visual symbol, attributes may correspond to its position, a zoom factor, a list of elementary graphical objects, a bitmap file, etc. The L component may be represented by a frame or a semantic network.

In this paper we mainly focus on the syntactic aspect. We consider three ways to syntactically represent a generalized sentence: *attribute-based*, *relation-based* and *linear*. We will see that each representation can be converted into the other, even though the conversions may not be 1-to-1.

Attribute-based Representation

In the attribute-based case, the sentence is represented by making explicit the *syntactic attributes* of the gsymbols composing it. In the case of the syntactic model String the attribute *position* of a gsymbol has as its value the position index of the gsymbol in the string; in the case of Plex, the attaching points of a gsymbol are numbered and represented by an array $ap[1], \dots, ap[n]$. The value of $ap[i]$ is given by a unique label assigned to the link plugged to attaching point i of the gsymbol. In the case of the syntactic model Graph the attaching regions of a gsymbol are numbered and represented by an array $aps[1], \dots, aps[n]$ of sets. The value of $aps[i]$ is the set of labels of the links plugged to attaching region i . Figure 2.3 (a) shows the attribute-based representation of an activity diagram by considering the link labeling provided in Figure 2.3 (b).

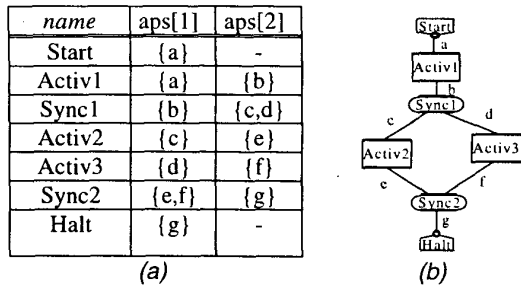


Figure 2.3 Attribute-based representation (a) of the activity diagram in Figure 2.1 (ii) based on the link labeling in (b)

Relation-based representation

Given a generalized sentence gs , let us consider a set R of binary relation identifiers. A labeled graph on R and gs , $G_{\langle R, gs \rangle} = (N, E)$, is defined as follows:

- each node in N identifies a distinct gsymbol in the sentence gs
- a labeled edge (x, y, REL) is in E iff $REL \in R$ holds between subsets of syntactic attributes of the generalized symbols x and y , respectively.

Definition 1. Let R and gs be a set of binary relation identifiers and a generalized sentence, respectively, a *relation-based representation* of gs with respect to R is any labeled graph $G_{\langle R, gs \rangle}$ that is connected. ♦

In the following, we will denote a labeled graph $G_{\langle R, gs \rangle}$ by listing its labeled edges in the format $REL(x, y)$.

As an example, let us consider the generalized sentence in Figure 2.3 (b). It can be modeled according to the syntactic model Graph, by using a class of relations of type $LINK_{i,j}$ so defined: a gsymbol x is in relation $LINK_{i,j}$ with a gsymbol y iff attaching point i of x is connected to attaching point j of y , i.e., iff $aps_x[i] \cap aps_y[j]$ is not empty. Under these premises, the relation-based representation of the activity diagram in Figure 2.3 is given by the set: $\{LINK_{1,1}(Start, Activ1), LINK_{2,1}(Activ1, Sync1), LINK_{2,1}(Sync1, Activ2), LINK_{2,1}(Sync1, Activ3), LINK_{2,1}(Activ2, Sync2), LINK_{2,1}(Activ3, Sync2), LINK_{2,1}(Sync2, Halt)\}$

Linear representation

Definition 2. Given a generalized sentence $gs = \{x_1, \dots, x_n\}$ and a set of relation identifiers R , a *linear representation* of gs with respect to R is the pair $(G_{\langle R, gs \rangle}, P)$ where:

1. $G_{\langle R, gs \rangle}$ is a relation-based representation of gs ;
2. P is a permutation (y_1, y_2, \dots, y_n) of the symbols in gs such that for each y_i with $1 \leq i \leq n$, there exists at least an index k such that $1 \leq k < i$ and an edge in $G_{\langle R, gs \rangle}$ on y_k and y_i . ♦

A linear representation $(G_{\langle R, gs \rangle}, P)$ will be denoted in the following as the string: $y_1 R_1 y_2 R_2 y_3 \dots R_{n-1} y_n$

where each R_i is a non-empty sequence of type:

$$\langle REL_1^{h_1}, \dots, REL_i^{h_i}, \dots, REL_m^{h_m} \rangle \quad \text{with } m \geq 1.$$

Each $REL_i^{h_i}$ denotes the pair (REL_i, h_i) where $REL_i \in R$ labels the edge (y_{j-h_i}, y_{j+1}) in $G_{\langle R, gs \rangle}$ and relates syntactic attributes of y_{j+1} with syntactic attributes of y_{j-h_i} , with $0 \leq h_i < j$. In the rest of the paper, we will denote REL_i^0 simply as REL_i .

Notice that, since $G_{\langle R, gs \rangle}$ is connected, it is always possible to find a linear representation for gs .

As an example, let us consider the activity diagram in Figure 2.3, if we consider its relation-based representation given above and the following permutation of symbols: (Start, Predicate, Statement, Statement', Halt) the linear representation is:

Start $\langle LINK_{1,1} \rangle$ Activ1 $\langle LINK_{2,1} \rangle$ Sync1 $\langle LINK_{2,1} \rangle$ Activ2 $\langle LINK_{2,1}^{-1} \rangle$ Activ3 $\langle LINK_{2,1}^{-1} \rangle$ Sync2 $\langle LINK_{2,1}^{-1} \rangle$ Sync2 $\langle LINK_{2,1} \rangle$ Halt.

This linear representation well fits the interpretation of an activity diagram. It follows the natural flow of the described activities. The same cannot be said if an alternative linear representation starting from Halt is

chosen. Thus, the semantics of a generalized sentence can drive the construction of one of its linear representations.

3. Extended Positional Grammars

Extended Positional Grammars (XPGs) are a direct extension of Positional Grammars (PGs) [2]. In particular, an Extended Positional Grammar is the pair (G, PE) , where PE is a *positional evaluator*, and G can be seen as a particular type of context-free¹ string attributed grammar $(N, T \cup POS, S, P)$ where:

- N is a finite non-empty set of *non-terminal* gsymbols;
- T is a finite non-empty set of *terminal* gsymbols, with $N \cap T = \emptyset$;
- POS is a finite set of *binary relation* identifiers, with $POS \cap N = \emptyset$ and $POS \cap T = \emptyset$;
- $S \in N$ denotes the starting gsymbol;
- P is a finite non-empty set of *productions* of the following format:

$$A \rightarrow x_1 R_1 x_2 R_2 \dots x_{m-1} R_{m-1} x_m \Delta, \Gamma$$

where A is a non-terminal gsymbol, $x_1 R_1 x_2 R_2 \dots x_{m-1} R_{m-1} x_m$ is a linear representation with respect to POS where each x_i is a gsymbol in $N \cup T$ and each R_j is partitioned in two sub-sequences

$$(\langle REL_1^{h1}, \dots, REL_i^{hi} \rangle, \langle REL_{i+1}^{hi+1}, \dots, REL_n^{hn} \rangle)$$

The relation identifiers in the first sub-sequence of an R_j are called *driver relations*, whereas the ones in the second sub-sequence are called *tester relations*. Driver relations are used during syntax analysis to determine the next gsymbol to be scanned, whereas tester relations are used to check whether the last scanned gsymbol (terminal or non-terminal) is properly related to previously scanned gsymbols.

Without loss of generality we assume that there are no useless symbols, and no unit and empty productions [1].

Δ is a set of rules used to synthesize the values of the syntactic attributes of A from those of x_1, x_2, \dots, x_m ;

Γ is a set of triples $\{(T_j, Cond_j, \Delta_j)\}_{j=1..k}$, $k \geq 0$, used to dynamically insert new terminal symbols in the input generalized sentence during the parsing process. In particular,

- ♦ T_j is a terminal gsymbol to be inserted in the input generalized sentence;
- ♦ $Cond_j$ is a pre-condition to be verified in order to insert T_j ;
- ♦ Δ_j is the rule used to compute the values of the syntactic attributes of T_j from those of x_1, \dots, x_m .

Moreover, a production $A \rightarrow x_1 \dots x_m \Delta, \Gamma$ must satisfy the following property:

"the number of triples in Γ whose conditions can simultaneously evaluate to true must be less than $m-1$ ".

¹ Here "context-free" means that the grammar productions are in "context-free" format and does not refer to the computational power of the formalism. This issue is not intended to be treated in this paper.

This means that no more than $m-2$ gsymbols can be inserted in the input during the production application. In particular, this guarantees the convergence of parsing algorithms based on XPGs.

Informally, a Positional Evaluator PE is a materialization function which transforms a linear representation into the corresponding generalized sentence in *external* representation.

In the following we characterize the languages described by an extended positional grammar $XPG = ((N, T \cup POS, S, P), PE)$.

We write $\alpha \Leftarrow \beta$ and say that β reduces to α in one step, if there exist δ, π, A, η such that

1. $A \rightarrow \eta, \Delta, \Gamma$ is a production in P ,
2. $\beta = \delta \eta \pi$,
3. $\alpha = \delta A' \gamma \pi$, where A' is a gsymbol whose attributes are set according to the rule Δ and γ results from the application of Γ .

We write $\alpha * \Leftarrow \beta$ and say that β reduces to α , if there exist $\alpha_0, \alpha_1, \dots, \alpha_m$ ($m \geq 0$) such that

$$\alpha = \alpha_0 \Leftarrow \alpha_1 \Leftarrow \dots \Leftarrow \alpha_m = \beta$$

The sequence $\alpha_m, \alpha_{m-1}, \dots, \alpha_0$ is called a *derivation* of α from β .

- a *positional sentential form* from S is a string β such that $S * \Leftarrow \beta$
- a *positional sentence* from S is a string β containing no non-terminals and such that $S * \Leftarrow \beta$
- a *generalized sentential form* (*generalized sentence*, resp.) from S is the result of evaluating a positional sentential form (positional sentence, resp.) from S through PE .

The *language described by an XPG*, $L(XPG)$, is the set of the generalized sentences from the starting symbol of XPG .

The new features of Extended Positional Grammars, as opposed to Positional Grammars, include the use of *multiple* driver relations and the introduction of Γ rules to dynamically modify the input generalized sentence. In particular, the Γ rules add context to productions. This mechanism is similar to the *remote* nodes of Picture Layout Grammars (PLG, [8]), to the *existentially quantified* symbols of Constraint Multiset Grammars (CMG, [10]) and to the *shared* symbols in [5]. The main difference with [8] and [5] is that Γ rules allow the use of contextual symbols possibly not present in the production. In all the cases, Γ rules set conditions on the use of contextual symbols. The use of context is motivated by the need for positional grammars to define the syntax of complex visual modeling languages such as the UML Class and Use Case diagrams, and Statecharts.

In the following we give an example of XPG for modeling State Transition Diagrams (STD in short) describing Finite State Automata.

Example 3.1. The $XPG = ((N, T \cup POS, S, P), PE)$ for STD describing Finite State Automata has the following

characteristics. The set of non-terminals is given by $N = \{\text{STDDiagram}, \text{Graph}, \text{Node}\}$ where each gsymbol has one attaching region as syntactic attribute.

The set of terminals is given by $T = \{\text{NODEI}, \text{NODEIF}, \text{NODEF}, \text{NODEG}, \text{EDGE}, \text{PLACEHOLD}\}$. The terminal gsymbols NODEI , NODEIF , NODEF , NODEG have one attaching region as syntactic attribute. They represent, respectively, the *initial*, *initial and final*, *final*, and *generic* nodes of a STD. The terminal gsymbol EDGE has two attaching points as syntactic attributes corresponding to the start and end points of the edge. Finally, PLACEHOLD is a fictitious terminal gsymbol to be dynamically inserted in the input sentence during the parsing process. It has one attaching region as syntactic attribute.

The tokens are graphically depicted in Figure 3.1. Here, each attaching region is represented by a bold line and identified by the number 1 while the two attaching points of EDGE are represented by bullets and identified by a number each. In the following, the notation Gsym_i denotes the attaching point i of the gsymbol Gsym .

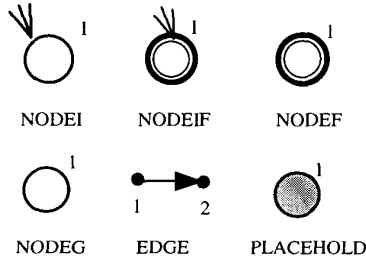


Figure 3.1. The terminals for the grammar STD

The set of relations is given by $\text{POS} = \{\text{LINK}_{h,k}, \text{any}\}$, where the relation identifier *any* denotes a relation that is always satisfied between any two gsymbols, whereas $\text{LINK}_{h,k}$ is as defined in Section 2.3 and will be denoted as h_k to simplify the notation. Moreover, we use the notation $\overline{h_k}$ when describing the absence of a connection between two attaching areas h and k .

Next, we provide the set of productions for describing STDs. (Notice that $\text{Graph}_1 = \text{Graph}'_1 - \text{EDGE}_1$ indicates set difference and is to be interpreted as follows: "the attaching area 1 of Graph is to be connected to whatever is attached to the attaching area 1 of Graph' except for the attaching point 1 of EDGE ". Moreover the notation $|\text{Node}_1|$ indicates the number of connections to the attaching area 1 of Node .)

- (1) $\text{STDDiagram} \rightarrow \text{Graph}$
- (2) $\text{Graph} \rightarrow \text{NODEI}$
 $\Delta: (\text{Graph}_1 = \text{NODEI}_1)$
- (3) $\text{Graph} \rightarrow \text{NODEIF}$
 $\Delta: (\text{Graph}_1 = \text{NODEIF}_1);$
- (4) $\text{Graph} \rightarrow \text{Graph}' \langle \langle \text{I}_1 \rangle, \langle \overline{\text{I}_2} \rangle \rangle \text{EDGE } 2_1 \text{ Node}$

- $\Delta: (\text{Graph}_1 = \text{Graph}'_1 - \text{EDGE}_1),$
 $\Gamma: \{ (\text{PLACEHOLD}; |\text{Node}_1| > 1;$
 $\text{PLACEHOLD}_1 = \text{Node}_1 - \text{EDGE}_2) \}$
- (5) $\text{Graph} \rightarrow \text{Graph}' \langle \langle \text{I}_1 \rangle, \langle \text{I}_2 \rangle \rangle \text{EDGE}$
 $\Delta: (\text{Graph}_1 = (\text{Graph}'_1 - \text{EDGE}_1) - \text{EDGE}_2);$
- (6) $\text{Graph} \rightarrow \text{Graph}' \langle \langle \text{I}_2 \rangle, \langle \overline{\text{I}_1} \rangle \rangle \text{EDGE } \text{I}_1 \text{ Node}$
 $\Delta: (\text{Graph}_1 = \text{Graph}'_1 - \text{EDGE}_2),$
 $\Gamma: \{ (\text{PLACEHOLD}; |\text{Node}_1| > 1;$
 $\text{PLACEHOLD}_1 = \text{Node}_1 - \text{EDGE}_1) \}$
- (7) $\text{Graph} \rightarrow \text{Graph}' \langle \text{any} \rangle \text{PLACEHOLD}$
 $\Delta: (\text{Graph}_1 = \text{PLACEHOLD}_1);$
- (8) $\text{Node} \rightarrow \text{NODEG}$
 $\Delta: (\text{Node}_1 = \text{NODEG}_1);$
- (9) $\text{Node} \rightarrow \text{NODEF}$
 $\Delta: (\text{Node}_1 = \text{NODEF}_1);$
- (10) $\text{Node} \rightarrow \text{PLACEHOLD}$
 $\Delta: (\text{Node}_1 = \text{PLACEHOLD}_1);$

According to these rules, a State Transition Diagram is described by a graph (production 1) defined as

- ♦ an initial node (production 2) or as
- ♦ an initial-final node (production 3) or, recursively, as
- ♦ a graph connected to a node through an outgoing (production 4) or incoming (production 6) edge, or as
- ♦ a graph with a loop edge (production 5).

A node can be either a generic node (production 8) or a final node (production 9).

The need for productions 7 and 10 will be clarified in the following example.

Example 3.2. Figure 3.2 shows how to recognize a state transition diagram through the extended positional grammar STD shown above. In particular, Figure 3.2(a) shows the initial STD, two dashed ovals indicating the handles to be reduced, and the corresponding productions to be used in the reductions. By applying production 2, the initial node NODEI is reduced to the non-terminal Graph . Due to the Δ rule of production 2, Graph inherits all the connections of NODEI . Similarly, the application of production 8 substitutes the circled NODEG in Figure 3.2(a) with the non-terminal Node . Figure 3.2(b) shows the resulting generalized sentential form and the handle for the application of production 4. The gsymbols Graph , EDGE and Node are then reduced to the new non-terminal Graph . Due to the Δ rule of production 4, the new Graph is connected to all the remaining edges attached to the old Graph . Moreover, due to the Γ rule, since $|\text{Node}_1| = 4 > 1$, a new node PLACEHOLD is inserted in the input and connected to all the remaining edges attached to the old Node . Figure 3.2(c) shows the resulting generalized sentential form.

After the application of productions 9 and 4 the generalized sentential form reduces to the one shown in Figure 3.2(e). From here, production 7 reduces the non-terminal Graph and PLACEHOLD to a new non-terminal

Graph. By applying the Δ rule of production 7, the new Graph inherits all the connections to PLACEHOLD (see Figure 3.2(f)). The subsequent application of productions 10, 5, 4 and 1 reduces the original state transition diagram to the starting gsymbol in Figure 3.2(i), determining the membership of the input to the language $L(STD)$.

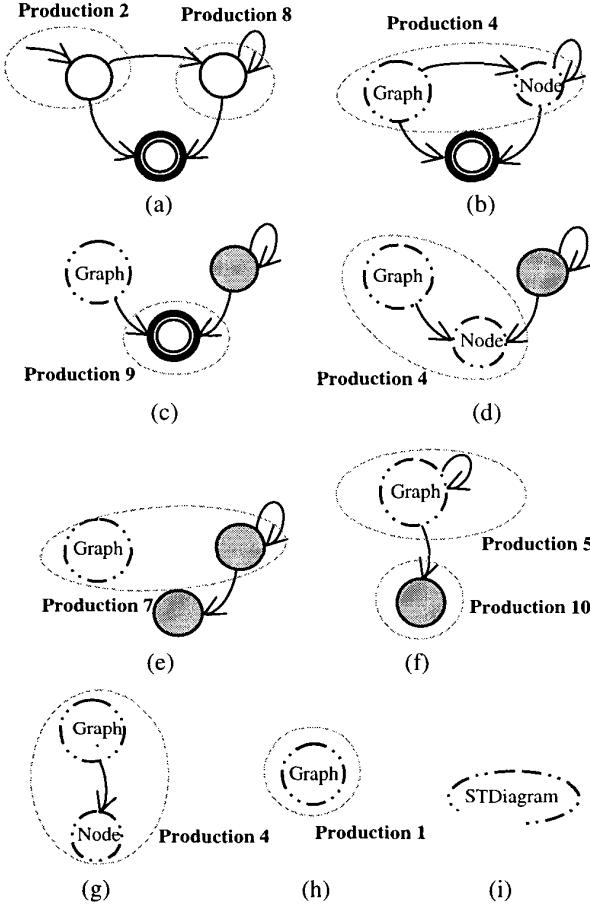


Figure 3.2. The reduction process for a STD

4. The XpLR Methodology

The XpLR methodology is an extension of the pLR methodology as presented in [2]. It is a framework for implementing visual systems based upon XPGs and LR parsing. As in pLR parsing, an XpLR parser scans the input in a non-sequential way, driven by the relations used in the grammar. In particular, the XpLR methodology differs from the pLR one in that it handles the Γ rules, and provides algorithms to eliminate conflicts arising during the construction of a pLR parsing table. These extensions allow the parsing of a wider class of visual languages than the one handled by the pLR methodology.

The components of an (X)pLR parser are shown in Figure 4.1 and are detailed in the following.

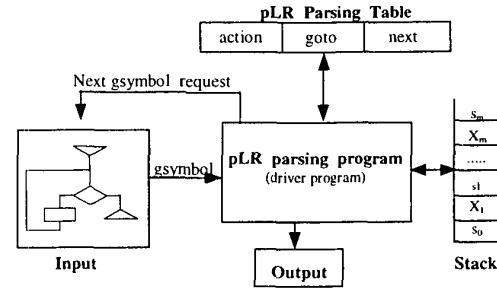


Figure 4.1 The architecture of an (X)pLR parser

The input

The input to the parser is a dictionary storing the relation-based representation of a picture as produced by the visual editor. No parsing order is defined on the graphical objects in the dictionary. The parser retrieves the objects in the dictionary by a *find* operation driven by the relations in the grammar. The parser implicitly builds and parses a linear representation from the input attribute-based representation.

The stack

An instance of the stack has the general format $s_0g_1s_1g_2s_2...g_ms_m$, where s_m is the stack top; g_i is a grammar gsymbol and s_i is a generic state of the parsing table. The parsing algorithm uses the state on the top of the stack and the gsymbol currently under examination to access a specific entry of the parsing table in order to decide the next action to execute.

The XpLR parsing table

An XpLR parsing table (see Figure 4.2) is composed of a set of rows and is divided in three main sections: *action*, *goto*, and *next*. Each row is composed of an ordered set of one or more sub-rows each corresponding to a parser state. The *action* and *goto* sections are similar to the ones used in LR parsing tables for string languages [1], while the *next* section is used by the parser to select the next gsymbol to be processed. An entry $next[k]$ for a state s_k contains the pair (R_{driver}, x) , which drives the parser in selecting the next gsymbol (derivable from x) by using the sequence of driver relations R_{driver} . The *action* and *goto* entries are named *conditioned actions* and have the format " $R_{tester}: state$ " and " $R_{tester}: shift state$ ", respectively, where R_{tester} is a possibly empty sequence of tester relations. A shift or goto action is executed only if all the relations in R_{tester} are true, or if R_{tester} is empty. As an example, let us consider the XpLR(0) parsing table in Figure 4.2. If the current state corresponds to sub-row 4.2 and the current gsymbol is EDGE, then the parser executes the conditioned action $(I_I: s7)$, and it goes to state s_7 only if the relation I_I does not hold between EDGE and the first gsymbol below the stack top. If it reaches state s_7 , due to $next[7] = (I_I, Node)$, it looks for a terminal derivable from Node which is in relation I_I with the just

St.	Action							Goto			NEXT
	NODEI	NODEIF	NODEF	NODEG	EDGE	PLACEHOLD	EOI	STDiagram	Graph	Node	
0	:s2	:s3						:1	:4		(start, STDiagram)
1							acc				(any, *)
2	r2	r2	r2	r2	r2	r2	r2				-
3	r3	r3	r3	r3	r3	r3	r3				-
4	1				$I_2: s5$						(I_1, EDGE)
	2				$I_2: s6$						(I_2, EDGE)
	3				$I_1: s7$						(any, PLACEHOLD)
	4	r1	r1	r1	r1	r1	r1				(any, *)
5			:s11	:s10		:s12				:9	(2_I, Node)
6	r5	r5	r5	r5	r5	r5	r5				-
7			:s11	:s10		:s12				:13	(I_1, Node)
8	r7	r7	r7	r7	r7	r7	r7				-
9	r4	r4	r4	r4	r4	r4	r4				-
10	r8	r8	r8	r8	r8	r8	r8				-
11	r9	r9	r9	r9	r9	r9	r9				-
12	r10	r10	r10	r10	r10	r10	r10				-
13	r6	r6	r6	r6	r6	r6	r6				-

Figure 4.2. An XpLR(0) parsing table

seen EDGE. It can be noted that, in this case, the parser is matching the input against production (6).

In general, other than the traditional *shift-reduce* and *reduce-reduce* conflicts, a pLR parsing table presents a *positional* conflict if there exists an entry of the *next* section containing more than one element, and a *shift-shift* conflict (*goto-goto* conflict, resp.) if the *action* section (*goto* section, resp.) presents an entry with more than one conditioned action with conditions that are not mutually exclusive, [2].

The XpLR methodology resolves the *shift-shift* and *reduce-reduce* conflicts by setting an ordering on the *conditioned actions* within the same entry of the parsing table. The XpLR parser checks the *conditions* in that order and executes the first action whose condition holds. A similar technique is used to solve *positional* conflicts: in this other case, the parser repeatedly invokes a function *Fetch_Gsymbol* on each pair (R, x), according to the defined order, until a new symbol is successfully returned or a syntactic error is detected.

All this is implemented by splitting a state in one or more ordered sub-states. In most cases this partitioning solves the conflicts and makes the language suitable for XpLR(0) parsing.

As an example, Figure 4.2 shows the XpLR(0) parsing table for the grammar STD. It can be noted that row 4 has been split in 4 ordered sub-states with no conflicts. In fact, even though entry (4.1, EDGE) presents two conditioned actions, these have mutually exclusive conditions. State 4 records the fact that the parser has just reduced a Graph and is ready to match, in this order, an outgoing EDGE with no loop (see state 4.1 and production 4), or an outgoing EDGE with loop (see state 4.1 and production 5), or an incoming EDGE (see state 4.2 and production 6), or any PLACEHOLD (see state 4.3 and production 7), or any End-Of-Input (EOI) (see state 4.4 and production 1). Note

that, the execution of the function *Fetch_Gsymbol* on the special *next* entry (*any, **) returns any terminal in the input or an EOI token in case there are no terminals left.

The XpLR parser

The additional features to be introduced in the XpLR parser with respect to the pLR parser presented in [2] regard the management of the multiple driver relations, the Γ rules and the management of the new format of parsing table. Multiple driver relations are handled in the function *Fetch_Gsymbol* that is called by the parser to fetch the next symbol to be processed. The Γ rules are handled during the handling of the "reduce" actions while the sub-states are taken care of during the "goto" and "shift" action execution. The new characteristics do not affect the function *Test* invoked by the driver to verify the action conditions in the *action* and *goto* sections of the parsing table. In the following, we give the complete XpLR(0) parsing algorithm.

Algorithm 4.1. The XpLR(0) parsing algorithm.

Input: A generalized sentence with n gsymbols in attribute-based representation and an XpLR(0) parsing table with no conflicts.

Output: A bottom-up analysis of the generalized sentence if this is syntactically correct, an error message otherwise.

Method: Start with the state s_0 on the top of the stack.

repeat forever

let s be the state on the stack top

if s contains a substate sb such that $next[sb]$ is not empty

then

if there exists at least a substate sb of s such that $Fetch_Gsymbol(next[sb])$ is not null then

let fsb be the first of such substates in s

set $s = fsb$ and $ip = Fetch_Gsymbol(next[fsb])$

```

endif
else emit "syntax error"; exit
let  $b$  the grammar symbol pointed by  $ip$ 
if action  $[s, b]$  is a non empty sequence  $seq$  of conditioned
    shifts of type " $R_i$ : shift  $s'$ " then
    while  $seq$  is not empty do
        extract the current first element " $R_i$ : shift  $s'$ " from  $seq$ 
        if  $R_i$  is empty or  $Test(REL^h, b)$  is true for each
             $REL^h \in R_i$  then
                push  $b$  and then  $s'$  on the stack and exit from the
                while loop
        endwhile
    if no element has been pushed on the stack then
        emit "syntax error" and exit;
    endif
elseif action  $[s, REDUCE] =$ 
    reduce  $A \rightarrow x_1 R_1 x_2 R_2 \dots R_{m-1} x_m, \Delta, \Gamma$  then
        compute the syntactic attributes of the gsymbol  $A$ 
        according to the synthesis rule  $\Delta$ , apply rule  $\Gamma$ , if
        present, and pop  $2*m$  symbols from the stack
    let  $s'$  be the new state on stack top,
    if goto $[s', A]$  is a non empty sequence  $seq$  of conditioned
        gotos of type " $R_i$ :  $s'$ " then
        while  $seq$  is not empty do
            extract the current first element " $R_i$ :  $s'$ " from  $seq$ 
            if  $R_i$  is empty or  $Test(REL^h, A)$  is true for each
                 $REL^h \in R_i$  then
                    push  $A$  and then  $s'$  on the stack and output
                    the production  $A \rightarrow x_1 R_1 x_2 \dots R_{m-1} x_m, \Delta, \Gamma$ 
            endwhile
        if no element has been pushed on the stack then
            emit "syntax error" and exit;
        endif
    endelseif
elseif action  $[s, b] =$  "accept" then "success"
else emit "syntax error" and exit;
endrepeat

```

For a fixed grammar, the time complexity is $O(n(tq + tr + t_\Delta))$ where the parameters tq , tr , and t_Δ depend on the particular syntactic model adopted and are, respectively, the time complexities of Fetch_Gsymbol(), of a condition (testers or Γ conditions), and of a synthesis rule in Δ . The complexity is unchanged with respect to that of the pLR parser shown in [2]. In fact, the new features that may add complexity, i.e., the splitting of a state in ns substates and the limited insertion of new terminals during reductions, only add constant factors.

It is easy to reproduce the reduction process in Figure 3.2 by applying Algorithm 4.1 on the XpLR(0) parsing table in Figure 4.2.

5. Conclusions

We have presented the XpLR grammar formalism and a new LR-based technique to parse visual languages. As a

result it is possible to handle a wider class of visual languages with respect to the pLR methodology.

The XpLR methodology is being used in the latest version of VLCC [4] and has been exploited so far to produce prototypes for many practical visual environments including Workflow Management Languages and most of the Unified Modeling Language (UML) diagrammatic elements such as the Use-case, Collaboration, State, Class, Activity and Deployment Diagrams.

At this point of our research on LR-based parsing of visual languages, we feel that the XpLR methodology is able to capture in an effective way the syntax of most constructs of visual languages. Currently, we are investigating the integration of the XpLR methodology and the VLCC framework with ad-hoc components for constructing meta-CASE environments, multimedia software engineering tools, and visual security policy specification tools.

References

- [1] A.V.Aho, R. Sethi, and J.D. Ullman, "Compilers, principles, techniques and tools", Addison-Wesley, New York, 1985.
- [2] G. Costagliola, A. De Lucia, S. Orefice, G. Tortora, "A Parsing Methodology for the Implementation of Visual Systems", IEEE Trans. Soft. Eng. 23(12), 1997, pp. 777-799.
- [3] G. Costagliola, A. De Lucia, S. Orefice, G. Tortora, "A Framework of Syntactic Models for the Implementation of Visual Languages," Procs. 13th IEEE Symposium on Visual Languages, Capri, Italy, September, 1997, pp. 58-65.
- [4] G. Costagliola, F. Ferrucci, G. Polese, and G. Vitiello, "Supporting Hybrid and Hierarchical Visual Language Definition", Procs. of 15th IEEE Symposium on Visual Languages, Tokyo, Japan, September, 1999.
- [5] G. Costagliola, A. De Lucia, S. Orefice, G. Tortora, "Efficient Parsing of Data-Flow Graphs", in Procs. of the 7-th Conference on Software Engineering and Knowledge Engineering, Rockville, MD, USA, June 22-24, 1995, pp. 226-233.
- [6] J. Feder, "Plex Languages", Information Science, vol. 3, 1971, pp. 225-241.
- [7] F. Ferrucci, G. Pacini, G. Satta, M. Sessa, G. Tortora, M. Tucci, and G. Vitiello, "Symbol-Relation Grammars: A Formalism for Graphical Languages", Information and Computation, vol. 131, November 1996, pp. 1-46.
- [8] E.J. Golin, "Parsing Visual Languages with Picture Layout Grammars", Journal of Visual Languages and Computing, 2, 1991, pp. 1-23.
- [9] K. Marriott and B. Meyer, (Eds.) "Visual Language Theory", Springer-Verlag, New York Inc., 1998.
- [10] K. Marriott, "Constraint Multiset Grammars", Procs of the 1994 IEEE Symposium on Visual Languages, pp. 118-125.
- [11] J. Rekers and A. Schurr, "A Graph Based Framework for the Implementation of Visual Environments", Procs 12th IEEE International Symposium on Visual Languages, Boulder, Colorado, Sept. 1996, pp. 148-157.
- [12] M. Tucci, G. Vitiello, G. Costagliola, "Parsing Non-linear Languages," IEEE Trans. Soft. Eng. 20(9), 1994, pp. 720-739.
- [13] K. Wittenburg, "Earley-style parsing for Relational Grammars", Procs 8th IEEE International Workshop on Visual Languages, Seattle, WA, U.S.A., 1992, pp. 192-199.