A Parsing Methodology for the Implementation of Visual Systems

Gennaro Costagliola, Member, IEEE, Andrea De Lucia, Sergio Orefice, and Genoveffa Tortora, Senior Member, IEEE

Abstract—The Visual Language Compiler-Compiler (VLCC) is a grammar-based graphical system for the automatic generation of visual programming environments. In this paper the theoretical and algorithmic issues of VLCC are discussed in detail. The parsing methodology we present is based on the “positional grammar” model. Positional grammars naturally extend context-free grammars by considering new relations in addition to string concatenation. Thanks to this, most of the results from LR parsing can be extended to the positional grammars inheriting the well-known LR technique efficiency. In particular, we provide algorithms to implement a YACC-like tool embedded in the VLCC system for automatic compiler generation of visual languages described by positional grammars.

Index Terms—Visual programming environments, multidimensional grammars, visual parser generation, pLR parsing, flowgraph languages.

1 INTRODUCTION

TTHE widespread use of visual systems has motivated the need for tools to support designers in the definition and implementation of visual language environments [2], [3], [9], [14], [27]. To this end, the Visual Language Compiler-Compiler (VLCC) system has been proposed in [7]. VLCC is a grammar-based graphical system that inherits, and extends to the visual field, concepts and techniques of compiler generation tools like YACC [18]. It allows to efficiently implement not only pure visual languages but also heterogeneous languages, i.e., languages integrating visual and textual notations, [7], [11]. In fact, text is useful for enhancing the expressiveness of visual languages as, for example, in the case of flowcharts. The VLCC system supports the implementation of a visual language by assisting the language designer during the definition of the graphical objects, the syntax and the semantics of the language. The generation process leads to an integrated environment comprising a graphical editor and a compiler for the defined visual language.

In [7], the VLCC system is shown from a user’s point of view; in this paper we focus on the underlying grammar formalism and the parsing algorithms. The grammar formalism underlying the VLCC system is the Positional Grammar model. Positional grammars naturally extend context-free grammars by considering new relations in addition to string concatenation. Thanks to this, it has been possible to exploit the LR methodology [1], to obtain an efficient LR-like parser for positional grammars (positional LR parser or pLR parser for short).

The positional LR methodology consists of algorithms to encode a positional grammar into a pLR parsing table. A shift-reduce syntax analysis deterministically driven by the parsing table is then performed. In order to guide the scanning of the input graphical objects, a new column is added to the usual action and goto parts of an LR parsing table. For each state of the parser, this column contains an entry with information to access the next object to be parsed.

A number of grammar formalisms to describe and parse visual languages have been introduced [4], [10], [15], [16], [17], [19], [21], [23], [24], [25], [26]. In general, the broader the class of languages to be treated is, the less efficient the parsing algorithm is. Due to this, big efforts are being made to characterize a class which is expressive enough and, at the same time, efficient to parse.

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 [15], Relation Grammars [10], and Constraint Multiset Grammars [21] fall into this class.

In the formalism of Picture Layout Grammars (PLGs) [15] a visual sentence is an unordered collection, namely a multiset of visual symbols with attributes containing positional information about symbols. Each production of a PLG has associated a set of semantic functions and constraints. The semantic functions specify how the values of the attributes of the symbol on the left-hand side of the production are synthesized in terms of the attribute values of the symbols on the right-hand side. The constraints are predicates over the attribute values of the right-hand side indicating when a production can be applied.

Marriott [21] has introduced a constraint-based formalism, called Constraint Multiset Grammars (CMGs), which is closely related to PLGs. A nonterminal symbol in a multiset
can be rewritten by a production in the grammar whenever the attributes of the symbols in the multiset satisfy a given constraint, which describes relationships between pictures. The main difference between PLGs and CMGs is that the latter formalism supports the specification of constraints over existentially quantified symbols. In other words, a production can specify constraints over the attributes of any symbol in the current sentential form. Another difference is that negative constraints are allowed in CMGs. Such constraints are used to test the nonexistence of subdiagrams in the pictures.

In the formalism of Relation Grammars (RGs) [10], [24] the productions contain a multiset of symbols on the right-hand side and a set of constraints to determine the validity of a production. In a Relation Grammar the constraints are expressed as evaluation rules written in a Prolog-like notation, rather than being used to compute attributes of aggregate objects as in PLGs and CMGs. An O(nlogn) parser has been constructed for a constraint-based subclass of RGs, called RG/1 grammars [13].

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 and the one presented by Wittenburg [26], have also been defined. A predictive pictorial parser processes the input objects according to an ordering criterion specified in the grammar. In [26], Wittenburg has proposed a predictive, Earley-style parsing algorithm for a subclass of relational grammars. The parser is suited to a class of nondeterministic languages whose relations are partial orders.

Recently, Lutz [19], [20], and Wills [25] have provided parsers for special cases of plex languages [12], called flowgraph languages suitable for describing data- and control-flow graphs of programs. Both techniques are based on chart parsing. In particular, Wills faces the problem of recognizing commonly used data structures and algorithms, named clichés, in a program. To do this, programs are translated into data-flow graphs and clichés are encoded as flow graph productions. The recognition is then reduced to a graph parsing problem. Because of the problem complexity, the resulting parser is exponential. On the other hand, Lutz treats a restricted class of flow graph languages for which he proposes a chart parsing algorithm that runs in polynomial time with respect to the number of tie-point relationships in the input graph, for a fixed grammar.

The main difference between positional grammars and the above formalisms is in the structure of the right-hand side of productions. In positional grammars the right-hand side of a production has a string-like form, while in all the other formalisms it is composed by a set of objects and a set of relations over them. As a consequence, positional grammars fix an order on the objects of a picture that is then used by the parsing process. The efficiency of the pLR parser is counterbalanced by the limits imposed on the positional grammars. However, the positional grammar model has been experimented within the VLCC system to successfully build parsers for a number of practical visual languages including two-dimensional arithmetical expressions, document layouts, chemical structures, logic diagrams, data-flow graphs, structured and semistructured flowcharts, electrical circuits, and finite automata graphical layouts.

The paper starts with a description of the VLCC system showing its global architecture and a sample application. After a formal definition of the syntax of visual languages, different types of visual languages are characterized depending on the structure of their graphical objects and on the way they can be composed. Next, the positional grammar formalism is presented together with the characterization of classes of positional grammars for different types of visual languages; as an example, the specification of flow graph languages like data-flow graphs and structured flowcharts is given. The positional LR methodology is then illustrated together with the algorithms for creating the pLR parsing table and the pLR parser. We also discuss time complexity considerations on the parsing algorithm and give some proofs for the applicability of the methodology. Moreover, positional grammars that present parsing conflicts are analyzed together with some heuristics to solve them. Finally, we discuss further research topics.

2 THE VLCC SYSTEM

VLCC [7] is a grammar-based graphical system for the automatic generation of visual programming environments. It supports the implementation of a visual language by assisting the language designer in defining the graphical objects, the syntax, and the semantics of the language. The final result of the generation process consists of an integrated environment comprising a visual editor and a compiler for the defined visual language.

To capture the widest range of visual languages, the VLCC system can be configured for a specific class of languages, e.g., diagrammatic, iconic, etc. A characterization of different language classes can be given depending on the structure of their graphical objects and on the way they can be composed. For example, an icon-oriented visual language is based on a set of primitive icons, characterized by the coordinates of a representative point (e.g., the centroid), and relations to spatially arrange those icons and form iconic sentences. Also, box-and-arrow diagrams are defined for primitive box objects with attaching points and for composition rules to join boxes at those attaching points through arrows.

A prototype of the VLCC system has been implemented using object-oriented technology under MSWindows 3.1[X]™ and MSWindows 95™. Fig. 1 shows the architecture of VLCC. The system is composed of two main modules:

- the Parametrized Grammar Editor (PGE)
- the Visual Programming Environment Generator (VPEG)

PGE is a parametric system that can be configured to support the creation and the manipulation of the graphical components for a given class C of visual languages. This specialization is obtained by providing the system with the set of data structures and functions, which implement basic operations over the items of the languages in the class C.
Once the specialization has been accomplished, PGE provides the language designer with visual editors to define specific visual grammars by directly and visually manipulating the syntax. Moreover, as a by-product, PGE derives the corresponding positional grammars in YACC-like format.

VPEG inputs the grammar specification produced by PGE and generates an integrated visual programming environment consisting of a compiler and of a visual editor for the defined language. Fig. 2 shows the functional diagram of VPEG. It is composed of two submodules: the Compiler Generator and the Editor Generator.

The Compiler Generator takes as input the positional grammar specification produced by the PGE module and automatically produces the C++ driver program for the compiler of the visual language, together with the parsing table. The Editor Generator inputs the graphical specification of the graphical objects and the relations of the grammar, and customizes a predefined visual editor template. Moreover, the Editor Generator integrates the compiler and the specialized visual editor to form a C++ source program whose compilation produces the final visual programming environment.

The output of the VLCC system is an integrated environment comprising a visual editor and a compiler for the defined visual language. The user inputs a picture through the visual editor by arranging on the screen the graphical objects defined in the visual language vocabulary. The visual editor translates the picture into an internal representation and stores it into a dictionary whose organization depends on the visual language class. The dictionary is then input to the compiler of the visual language.

Fig. 3 shows the visual programming environment generated by VLCC for the translation of flowcharts into their corresponding ALGOL-like programs.

3 VISUAL LANGUAGES

While string languages are composed of sentences obtained by concatenating textual symbols, visual languages are defined as sets of pictures obtained by spatially arranging graphical objects over a given vocabulary through composition rules. Then, a picture is described by a set of graphical objects and a set of spatial relations over them denoting the composition rules used.

The graphical objects of a visual language vocabulary are characterized by an image and a set of attributes, named syntactic attributes. A relation relates graphical objects through their syntactic attributes and may be graphically represented in the picture.

The picture in Fig. 4a is obtained composing icons by using horizontal and vertical concatenation, while the picture in Fig. 4b is the result of interconnecting boxes through polylines. Note that while the horizontal and vertical concatenations among icons have no explicit visualization, the interconnections among boxes are visually depicted as polylines. We will refer to the former type of relations as implicit relations and to the latter ones as explicit relations.

Each icon in Fig. 4a has as image either a square or a triangle, and as syntactic attribute the position in the Cartesian plane specified by the coordinates of its centroid. For example, the vertical concatenation relation holds between the positions of the triangle icons labeled “1” and “3.”

The relative representation of a picture is a string-like representation alternating graphical objects to relation identifiers. Such a representation sets an ordering on the graphical objects and defines binary relations between them. A possible relative representation of the picture in Fig. 4a is shown in Fig. 5. Here, the HOR and VER identifiers are used to express the horizontal and vertical concatenation relations, respectively.
The number $i$ appearing as superscript of a relation identifier, say $REL$, indicates that $REL$ holds between the $(i+1)$th preceding icon in the string and the next icon. For example, $VER^1$ vertically relates the triangle labeled “3” to the triangle labeled “1” (the second icon in the string before $VER^1$).

Note that more than one relative representation may correspond to a given picture depending on the ordering and the set of binary relations used in the representation. As an example, another relative representation of the picture in Fig. 4a is shown in Fig. 6. Here, the $DIA$ identifier is used to express the diagonal concatenation.

Unlike the relative representations, only one absolute representation can be given for a picture once the syntactic attributes of its graphical objects have been set.

Different visual language types can be characterized depending on the syntactic attributes of the graphical objects and on the composition rules. The iconic visual language of the previous example has icons as objects whose syntactic attributes are the coordinates of the centroid, and horizontal and vertical concatenations as composition rules.

Another sample type of visual language is given by the flow graph languages. A graphical object of a flow graph language is usually a box image and a set of syntactic at-
attributes specifying particular points on the edge of the image; these points are named attaching points. A joint relation is usually used to link graphical objects through their attaching points, and is visualized by a polyline. The picture in Fig. 4b is a flowchart resulting from the composition of the boxes start, function, predicate, and halt.

To provide absolute representations for flow-graphs, we label attaching points and polylines. Fig. 7 shows the flowchart of Fig. 4b, where the attaching points are labeled by numbers and the polylines by the letters a, b, and c.

The absolute representation of the flowchart of Fig. 7 is shown in Table 2. Here, the table entry (function, attaching point 1) = b means that the polyline b is connected to the attaching point 1 of the object function.

A relative representation of the flowchart above follows:

\[
\text{start} \text{JOINT}^0(1, 1) \text{ predicate JOINT}^0(2, 1), \text{JOINT}^0(1, 2) \text{ function JOINT}^3(3, 1) \text{ halt}
\]

Here, \text{JOINT}^i(3, 1) relates the attaching point 3 of the object predicate to the attaching point 1 of the object halt through the relation \text{JOINT}. Note that the composite relation \text{JOINT}^0(2, 1), \text{JOINT}^0(1, 2) indicates that both joint relations hold between predicate and function.

Even though the relative representation above presents only binary relations, in the flowchart of Fig. 7 the polyline labeled “a” represents a ternary relation among the boxes 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 can always be represented by \(n-1\) binary relations. For instance, the ternary relation labeled “a” in Fig. 7 is represented in the relative representation above by the following two binary relations:

- \text{JOINT}(1, 1) holding between start and predicate
- \text{JOINT}(1, 2) holding between predicate and function.

The traditional string languages can also be considered as a particular type of visual language. In fact, an object from a string language vocabulary is characterized by a textual symbol image and a syntactic attribute specifying its position in a string. The only relation used in string languages is the string concatenation relation.

### 4 Positional Grammars

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

**Definition 4.1.** A context-free positional grammar is a six-tuple

\[
\text{PG} = (N, T, S, P, \text{POS}, \text{PE}),
\]

where

- \(N\) is a finite nonempty set of nonterminals
- \(T\) is a finite nonempty set of terminals, with \(N \cap T = \emptyset\)
- \(S\) denotes the starting nonterminal
- \(P\) is a finite nonempty set of productions
- \(\text{POS}\) is a finite set of binary relation identifiers
- \(\text{PE}\) is a pictorial evaluator

Both terminals and nonterminals (grammar objects in the following) are graphical objects as defined in Section 3.

Each production in \(P\) has the form

\[
A \rightarrow x_1R_1x_2 \ldots R_{m-1}x_m, \Delta
\]

where \(A\) denotes a nonterminal, each \(x_i\) denotes a grammar object and each \(R_i\) is a sequence of the form \(\text{REL}^{i_1}, \ldots, \text{REL}^{i_j}, \ldots, \text{REL}^{i_n}\) with \(n \geq 1\). Each \(\text{REL}_{i_j}\) is a relation identifier in \(\text{POS}\) relating the values of the syntactic attributes of \(x_{j+1}\) with the ones of \(x_{j-1}\), \(0 \leq j < n\). In the following, we will denote \(\text{REL}_{i_j}\) simply as \(\text{REL}_i\). The first relation appearing in a sequence \(R_i\) is called driver relation, the other relations will be referred to as tester relations. \(\Delta\) is a rule which synthesizes the syntactic attribute values of \(A\) from those of \(x_1, x_2, \ldots, x_m\).

\(\text{PE}\) 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) \(\text{PG} = (N, T, S, P, \text{POS}, \text{PE})\).

We write \(\alpha \Rightarrow \beta\) and say that \(\beta\) is derived from \(\alpha\) in one step, if there exist \(\delta, \gamma, A, \eta\) such that \(\alpha = \gamma \Delta \delta, A \rightarrow \eta \Delta\) is a production in \(P\) and \(\beta = \gamma [\eta \Delta] \delta\). The square brackets maintain the precedence in the application of the pictorial evaluator \(\text{PE}\).

We write \(\alpha \Rightarrow^* \beta\) and say that \(\beta\) is derived from \(\alpha\) if there exist strings \(\alpha_{i_0}, \alpha_{i_1}, \ldots, \alpha_m (m \geq 0)\) such that \(\alpha = \alpha_{i_0} \Rightarrow \alpha_{i_1} \Rightarrow \ldots \Rightarrow \alpha_m = \beta\).

The sequence \(\alpha_{i_0}, \alpha_{i_1}, \ldots, \alpha_m\) is called a derivation of \(\beta\) from \(\alpha\).

Let \(x\) be a grammar object:

- a positional sentential form from \(x\) is a string \(\beta\) such that \(x \Rightarrow^* \beta\).
- a positional sentence from \(x\) is a positional sentential form from \(x\) which does not contain nonterminals.
• 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 obtains a picture from its relative representation.

**Example 4.1.** Let us consider the following positional grammar describing a very simple iconic language whose pictures are the vertical concatenation of sequences of squares starting with a triangle.

\[
N = \{S, A\} \\
T = \{\triangle, \Box\} \\
POS = \{HOR, VER\} \\
P = \{\}
\]

\[
S \rightarrow \triangle HOR A VER^i S' \\
\quad \Delta_1 = \{(\text{Head}(S), \text{Tail}(S)) := (\text{Head}(\triangle), \text{Tail}(S'))\}
\]

\[
S \rightarrow \triangle HOR A \\
\quad \Delta_2 = \{(\text{Head}(S), \text{Tail}(S)) := (\text{Head}(\triangle), \text{Tail}(A))\}
\]

\[
A \rightarrow \Box HOR A' \\
\quad \Delta_3 = \{(\text{Head}(A), \text{Tail}(A)) := (\text{Head}(\Box), \text{Tail}(A'))\}
\]

\[
A \rightarrow \Box \\
\quad \Delta_4 = \{(\text{Head}(A), \text{Tail}(A)) := (\text{Head}(\Box), \text{Tail}(\Box))\}
\]

Each nonterminal has two syntactic attributes, named Head and Tail, both specifying an \((x, y)\) pair of coordinates in the 2D Cartesian plane; each terminal icon has one syntactic attribute (the pair of coordinates of its centroid), referred to as Head or Tail interchangeably. In the production, superscripts are used to distinguish different occurrences of the same nonterminal.

The relations identified by HOR and VER are the usual horizontal and vertical concatenations in the 2D Cartesian plane and are defined as follows. Let \(x \) and \(y\) denote grammar objects; if \(\text{Tail}(x) = (i, j)\) and \(\text{Head}(y) = (k, h)\), then

\[
x \text{ HOR } y \quad \text{iff} \quad h = i + 1 \text{ and } k = j \text{ and } \]  
\[
x \text{ VER } y \quad \text{iff} \quad h = i \text{ and } k = j + 1
\]

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

\[
S \Rightarrow [\triangle \text{ HOR } A \text{ VER}^i S', \Delta_1] \\
\Rightarrow [\triangle \text{ HOR } A \text{ VER}^i [\triangle \text{ HOR } A, \Delta_2], \Delta_1] \\
\Rightarrow \cdots \\
\Rightarrow [\triangle \text{ HOR } [\Box, \Delta_3] \text{ VER}^i [\triangle \text{ HOR } [\Box, \Delta_4], \Delta_3], \Delta_2], \Delta_1]
\]

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 HOR and VER, if any, and applies the corresponding synthesizing rule \(\Delta\). Whenever \(X \text{ HOR } Y\) (or \(X \text{ VER } Y\), respectively) must be evaluated, PE spatially arranges \(X\) and \(Y\) such that the Tail of \(X\) is in horizontal (vertical, respectively) concatenation with the Head of \(Y\). Note that \(X\) and \(Y\) can be grammar objects or partial results of PE.

By applying PE to the sentential form \([\triangle \text{ HOR } A \text{ VER}^i [\triangle \text{ HOR } A, \Delta_2], \Delta_1]\) the following pictorial form is obtained:

\[
\triangle \quad A \\
\triangle \quad A
\]

The application of PE to the derived positional sentence from \(S\) yields the following picture:

\[
\triangle \Box \\
\triangle \Box
\]

Different classes of positional grammars can be characterized by properly specifying the structure of the grammar objects (i.e., the type of their syntactic attributes), the set of relations, the synthesizing rules, and the pictorial evaluator PE.

Once the syntactic attributes for each grammar object have been specified, a particular class of positional grammars is characterized; we will refer to this class as a PG_class. Positional grammars in the same PG_class share the structure of the grammar objects, the way the relations and the synthesizing rules are defined and a particular pictorial evaluator PE. As a matter of fact, each PG_class describes a specific class of visual languages, i.e., visual languages of the same type.

**Example 4.2.** The specification of the PG_class Icons for iconic visual languages follows.

1) Each nonterminal has two syntactic attributes, named Head and Tail, both specifying an \((x, y)\) pair of coordinates in the 2D Cartesian plane; each terminal icon has an icon as image and has only one syntactic attribute (i.e., the pair of coordinates of its centroid), referred to as Head or Tail interchangeably.

2) Each relation identifier REL in POS is characterized by a pair \((p, q)\) of integers, and represents a binary relation on grammar objects in the 2D Cartesian plane. Let \(x\) and \(y\) denote grammar objects, if \(\text{Tail}(x) = (i, j)\) and \(\text{Head}(y) = (k, h)\), then

\[
x \text{ REL } y \quad \text{iff} \quad k = i + p \text{ and } h = j + q
\]

Moreover, given a production \(A \rightarrow x_1 R x_2 \ldots R_{m–1} x_m\) \(\Delta\), each \(R_i\) is of the form REL\( \hat{i}\), where REL\( \hat{i}\) is a relation identifier in POS.

3) Given a production \(A \rightarrow x_1 R_{m–1} \cdots R_{1} x_m \Delta\), the synthesizing rule \(\Delta\) is specified as follows:
\[ \Delta = \{(\text{Head}(A), \text{Tail}(A)) : (\text{Head}(x_i), \text{Tail}(x_j)) \} \]

with \(1 \leq j \leq m\)

i.e., the left-hand side symbol \(A\) is assigned, as Head, the Head of \(x_i\) and, as Tail, the Tail of some symbol \(x_j\) in the right-hand side.

4) \(PE\) is a function specified as follows:

INPUT: a positional sentential form \(\Sigma\) from a grammar object \(x\)

OUTPUT: a picture \(\Pi\) from \(x\) recursively obtained as follows:

if \(\Sigma\) is the grammar object \(x\)

then \(\Pi\) is \(x\) itself with \(\text{Tail}(x) = \text{Head}(x) = \) the coordinates of the centroid of \(x\) once this has been arranged in the 2D Cartesian plane

else begin

let \(\Sigma = \Sigma_1 \text{ REL } \Sigma_2 \text{ REL } \cdots \text{ REL } \Sigma_{m-1} \Sigma_m \Delta\)

where each \(\Sigma_i\) is a positional sentential form from a grammar object \(x_i\)

if \(\Pi_i\) is the pictorial form from \(x_i\) (\(i = 1, \ldots, m\), respectively), obtained by applying \(PE\) on \(\Sigma_i\) (\(i = 1, \ldots, m\), respectively)

then \(\Pi\) is obtained by arranging \(\Pi_1, \Pi_2, \ldots, \Pi_m\) in the 2D Cartesian plane such that \(\text{Tail}(x_i) \text{ REL } \text{Head}(x_{i+1})\) for \(1 \leq i \leq m - 1\), and with \((\text{Head}(x), \text{Tail}(x)) = (\text{Head}(x_i), \text{Tail}(x_j)) (1 \leq j \leq m)\) as specified by \(\Delta\)

end

It is easy to verify that the positional grammar of Example 4.1 belongs to the PG_class \(\text{Icons}\). In that grammar, \(\text{HOR}\) and \(\text{VER}\) are characterized by the pairs \((1, 0)\) and \((0, -1)\), respectively.

5 DESCRIBING FLOW GRAPH LANGUAGES BY POSITIONAL GRAMMARS

Let us recall the definition of the PG_class \(\text{Flow Graph}\) [6] which describes a wide variety of diagrammatic languages; it is characterized as follows:

1) Terminals and nonterminals are graphical objects with syntactic attributes specifying particular points named attaching points (labeled by numbers). According to [12], we will refer to terminal and nonterminal graphical objects as terminal and nonterminal NAPEs (n attaching-point entities), respectively.

For example, the following NAPEs represent “AND,” “OR,” “NOT” gates, and a “Flip-Flop”:

Both the “AND” and “OR” gates have three attaching points identified by the numbers 1, 2, and 3; the “NOT” gate has only two attaching points (numbers 1 and 2). The Flip-Flop “FF” has four attaching points (numbers 1, 2, 3, and 4). NAPEs can be connected through their attaching points.

2) \(POS\) contains only one identifier, \(\text{JOINT}\), for a relation of the type \(\text{JOINT}(h, k)\) defined as follows:

given two NAPEs denoted by \(x\) and \(y\), the relation \(x \text{ JOINT}(h, k) y\) holds iff the attaching point \(h\) of \(x\) is connected to the attaching point \(k\) of \(y\). Moreover, given the production \(A \rightarrow x_1 R_1 x_2 \cdots R_j x_{j+1} \cdots R_{m-1} x_m \Delta\), if \(\text{JOINT}(h, k)\) occurs in \(R_j\), then the attaching point \(h\) of \(x_{j-1}\) is connected to the attaching point \(k\) of \(x_{j+1}\).

The following picture shows two NAPEs, say \(x\) and \(y\), representing the “OR” and “NOT” gates respectively, for which \(x \text{ JOINT}(3, 1) y\) holds:

In the following, \(\text{JOINT}(h, k)\) will be written as \(h \_ k\) if \(i \neq 0\) and as \(h \_ k\) if \(i = 0\).

3) For each production

\[ A \rightarrow x_1 R_1 x_2 \cdots R_{m-1} x_m \Delta \]

where \(A\) has \(n\) attaching points (1, ..., \(n\)), the synthesizing rule \(\Delta\) has the form

\[\{\text{(1, ..., } i, \ldots, n) \rightarrow \text{(q11 ... q1m, ..., qi1 ... qim, ..., qn1 ... qnm)}\}\]

The value of the \(i\)th attaching point of the nonterminal \(A\) is defined by a string \(q_{i1} \ldots q_{in}\) (tie-point in the following) of attaching point identifiers; each \(q_i\) 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, if \(m = 3\) the tie-point \(q_{10} q_{20} q_{30} = 200\) defines the attaching point \(i\) of \(A\) to be the attaching point 2 of \(x_i\).

The rule \(\Delta\) is actually the Tie rule used in [12].

Consider the following picture representing a sample combinatorial circuit:

It is described by the positional production:

\[ \text{Circuit} \rightarrow \text{OR } 1 \_ 3 \text{ AND } 2 \_ 2 \text{ NOT}, \{(1, 2, 3, 4) \equiv (010, 020, 001, 300)\} \]

where the dashed box is the graphical representation of the nonterminal Circuit with its four attaching points. The Tie rule in the production describes the
attaching points of the nonterminal Circuit. For example, the third component 001 defines the attaching point 3 of the Circuit nonterminal as the attaching point 1 of the "NOT" gate.

4) PE is a function that takes as input a positional sentential form \( \Sigma \) from a NAPE \( x \) and gives in output a pictorial form \( \Pi \) obtained as follows:

\[
\text{if } \Sigma = x \text{ is a NAPE then } \Pi \text{ is the NAPE } x \text{ itself}
\]

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

As an example, consider the following positional sentential form:

\[
[\text{FF } \hat{\text{3}}_1, 4_3 \text{ OR } 1_3 \text{ AND } 2^{1}_1 \text{ NOT, (010, 020, 001, 300)}] \\
(2_2, 4_1 \text{ AND, (100, 200, 022, 003))}
\]

The application of PE will be done in a bottom-up fashion by evaluating first the positional sentential form \([\text{OR } 1_3 \text{ AND } 2^{1}_1 \text{ NOT, (010, 020, 001, 300)}]\). As a final result, the following picture representing a sequential network is obtained:

Two examples of positional grammars belonging to the PG_class Flow Graph follow.

**Example 5.1.** The following positional grammar generates tree-structured data-flow graphs. The terminal NAPEs and the productions of the grammar are graphically depicted in Figs. 8 and 9, respectively.

\[
N = \{\text{DFL, IF}\}
\]

\[
T = \{\text{op, arg, test, mux}\}
\]

\[
S = \text{DFL}
\]

\[
\text{POS} = \{\text{JOINT}\}
\]

\[
P = \begin{align*}
1) & \text{DFL } \rightarrow \text{ arg } & (1) = (1) \\
2) & \text{DFL } \rightarrow \text{ op } 1_1 \text{ DFL } 2^{1}_1 \text{ DFL'} & (1) = (300)
\end{align*}
\]

3) \text{DFL } \rightarrow \text{ mux } 1_1 \text{ DFL } 2^{1}_1 \text{ DFL'} 3^{1}_1 \text{ IF } (1) = (4000)

4) \text{IF } \rightarrow \text{ test } 1_1 \text{ DFL } 2^{1}_1 \text{ DFL'} \text{ (1) = (3000)}
\]

Fig. 8. The terminal NAPEs of the data-flow graph grammar.

Fig. 9. The visual productions for the data-flow graph grammar.

The object \text{op} represents the operation node, where OP is an operator from the set \{+,'-','\!', '/', '*\}. The condition node is represented by the object \text{test}, where TEST is in \{‘>’, '<', '<=', '>', '==', '!='\}. Attaching points 1 and 2 of the objects \text{op} and \text{test} denote the inputs of the nodes, while attaching point 3 denotes the output. The object \text{arg} is the argument node for the constants and variables of a data-flow graph.
(for example, ARG = 1, 2, ‘a’, ‘b’, x, y, etc.), while mux describes the multiplexer node. Here, attaching points 1, 2, 3, and 4 identify, respectively, the input lines true and false, the select line, and the output line.

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.

Fig. 10a and 10b show a picture generated by the positional grammar above and its meaning, respectively.

**Example 5.2.** The following positional grammar generates structured flowcharts with loops. The terminal NAPEs and the productions of the grammar are graphically depicted in Figs. 11 and 12, respectively. Here, the terminal names start, halt, begin, end, function, and predicate are also part of the image of the corresponding terminals.

N = {F, C, S, R}
T = {start, halt, begin, end, function, predicate}
POS = {JOINT}
P = { 1) F → start 1_1 C 2_1 halt
2) C → begin 2_1 R (1, 2) = (10, 02)
3) R → S 2_1 R (1, 2) = (10, 02)
4) R → end (1, 2) = (1, 2)
5) S → function (1, 2) = (1, 2)
6) S → predicate (2_1, 1_2) C (1, 2) = (12, 30)
7) S → predicate 2_1 C (3_1, 2_2) C (1, 2) = (100, 022) }

The production 1) defines a flowchart F as a compound statement C delimited by the start and halt NAPEs. In production 2) a compound statement C is defined as a begin NAPE connected to a box R. Productions 3) and 4) define a box R as a sequence of statements S ending with the end NAPE. Productions 5), 6), and 7) describe a statement S as a simple function, a WHILE loop construct, and an IF THEN ELSE construct, respectively.
Note that the starting nonterminal \( F \) has no attaching points, while all the other nonterminals have two attaching points.

## 6 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]. Fig. 13 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 \( \text{next} \), to the LR parsing table. The column \( \text{next} \) associates a pair \((\text{REL}, x)\) to each parser state \( I_k \), where \( \text{REL} \) is a relation and \( x \) is a grammar object. Whenever the parser reaches state \( I_k \), the pair \((\text{REL}, x)\) 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 [2] for two-dimensional symbolic languages.

In the following, we describe the parser components in detail.

### 6.1 The Input

The input to the parser is a dictionary storing the absolute 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 \textit{find} operation driven by the relations contained in the column \( \text{next} \) of the parsing table. In this way, the parser implicitly builds a relative representation from the input absolute representation.

For the sake of parsing efficiency, the implementation of the dictionary is chosen so as to optimize the implementation of the \textit{find} operation. For example, in the case of the PG\_class \texttt{Icons}, the dictionary is implemented by a two-dimensional array where rows of pairs and column indexess correspond to the coordinates in the 2D Cartesian plane. The find operation is then implemented as a random access to the array.

If the input picture contains explicit relations, i.e., the relations have a graphical representation as in the case of labeled polylines used in the PG\_class \texttt{Flow Graph}, its absolute representation is augmented with an array COUNTER containing an entry for each explicit relation. The entry COUNTER\((r)\) for an explicit relation labeled \( r \) with degree \( n \) contains the value \( n - 1 \). As explained in Section 3, the value \( n - 1 \) indicates the number of binary relations describing \( r \) in any relative representation of the picture.

Fig. 14 shows the data-flow graph of Fig. 10 and its corresponding absolute representation \( D_p \). As the grammar uses the explicit relation \text{JOINT}, also the array COUNTER is provided.

During the parsing phase, all the tokens visited and the binary explicit relations traversed are marked in order to guarantee that each object and each explicit relation is considered at most once. The marking of an explicit binary relation \text{REL} labeled \( r \) is done by decreasing the entry COUNTER\((r)\) by 1.

The 0-entry of \( D_p \) always refers to the end-of-input symbol \$. Anologously 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., all the input tokens have been parsed and all the explicit relations have been traversed. These conditions are signaled by having all the tokens marked and COUNTER\((r) = 0\) for each explicit relation \( r \), respectively.

The pLR parser uses a stack-like data structure which behaves as a stack except that also elements below the top may 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 \( s_0X_1s_1X_2 \ldots X_msm \), 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.

### 6.2 The pLR Parsing Table

A positional LR (pLR) parsing table consists of three parts: the \textit{action} and \textit{goto} parts of the LR parsing tables and the \textit{next} part (see Fig. 15). The column \textit{next} is used to implement the
syntactic directed scanning of the input. Whenever the parser reaches state $I_k$, the pair $(REL, x)$ in the entry $next[1, j]$ (if defined) is used to select from $D_j$ the next object to be processed (by a shift action) 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]. In particular, 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.

- **Table 6.3 The pLR Parsing Program**

<table>
<thead>
<tr>
<th>state</th>
<th>action</th>
<th>goto</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\leq 2$</td>
<td>$\leq 3$</td>
<td>$\leq 4$</td>
</tr>
<tr>
<td>1</td>
<td>$\leq 3$</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
</tr>
<tr>
<td>2</td>
<td>$\leq 2$</td>
<td>$\leq 3$</td>
<td>$\leq 3$</td>
</tr>
<tr>
<td>3</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
</tr>
<tr>
<td>4</td>
<td>$\leq 2$</td>
<td>$\leq 3$</td>
<td>$\leq 3$</td>
</tr>
<tr>
<td>5</td>
<td>$\leq 2$</td>
<td>$\leq 3$</td>
<td>$\leq 3$</td>
</tr>
<tr>
<td>6</td>
<td>$\leq 2$</td>
<td>$\leq 3$</td>
<td>$\leq 3$</td>
</tr>
<tr>
<td>7</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
</tr>
<tr>
<td>8</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
</tr>
<tr>
<td>9</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
</tr>
<tr>
<td>10</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
</tr>
<tr>
<td>11</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
</tr>
<tr>
<td>12</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
<td>$\leq 2$</td>
</tr>
</tbody>
</table>

Fig. 15. An example of pLR parsing table.

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 \( (REL_1^{h_1}, \ldots, REL_n^{h_n}) \) appears in the right-hand side of a production of a positional grammar, the driver relation $REL^{h_1}$ 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 \( (REL_2^{h_2}, \ldots, REL_n^{h_n}) \) will be used as action condition in the goto or action part of the table.

To better explain the use of the action conditions, let us assume that “$REL$: shift $s$” is an entry in the parsing table for a terminal $a$. Then, the shift operation can be performed if and only if $a$ is the current terminal and it is in relation $REL$ with the (i + 1)th object below the stack top. Analogously, if “$REL$: $s$” is an entry in the goto part for a non-terminal $A$, then the goto operation on $A$ can be performed if and only if $A$ is in relation $REL$ with the (i + 1)th grammar object below the stack top.

As an example, Fig. 15 shows the pLR parsing table for the positional grammar of Example 5.1. 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 $r_1$, 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 nonterminal 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 above, the identifier $SP$ refers to the node in the data-flow graph producing the final output. For example, in the data-flow graph of Fig. 10 this node is the node $mux$.

### 6.3 The pLR Parsing Program

In order to illustrate how the pLR parsing program works, three functions $GFIRST$, $Positional_Query$ and $Test$ must be defined. The function $GFIRST$ is a generalization of the function $FIRST$ [1]. It takes as input a pair $(x, k)$, where $x$ is a grammar object and $k$ denotes a syntactic attribute of $x$, and returns a set of couples $(y, h)$, where $y$ is a terminal, such that $y$ begins a positional sentence from $x$, and the syntactic attribute $k$ of $x$ can be synthesized from the syntactic attribute $h$ of $y$. The function $Positional_Query$ corresponds to the find operation on the dictionary. It uses $GFIRST$ to retrieve the row index in $D_j$ of the next object to be parsed. $Positional_Query$ uses the stack and the input as global data structures and takes its arguments from the column $next$ of the parsing table. The function $Test$ returns a Boolean value and is used to validate the tester relations between objects. It takes as input an action condition from the action or goto part of the parsing table.

The definition of the three functions depends on the class of positional grammars. In the following, we give algorithms for the three functions in the general case and show their specialization for the PG Class Flow Graph. Without loss of generality, it is assumed that PG has no empty productions.

**function $GFIRST(x, k)$**

\[ \forall x \text{ a grammar object, } k \text{ is one of its syntactic attributes} \]

**begin**

if $x$ is a terminal then let $GFIRST(x, k) = \{(x, k)\}$

if $x$ is a nonterminal and $x \rightarrow x_1 R_1 x_2 \ldots x_m R_m x_m$ is a production in PG where the attribute $k$ of $x$ is synthesized from the attribute $b$ of $x_i$ in $\Delta$

then for each $(a, h)$ in $GFIRST(x_1, b)$ do $add (a, h)$ to $GFIRST(x, k)$

end

Note that this function may be specialized to any class of positional grammars, by specifying the syntactic attributes and the rule $\Delta$. For the PG Class Flow Graph the syntactic
attributes are attaching points and the sentence

“where the attribute k of x is synthesized from the
attribute b of x₁ in Δ”

is substituted by:

“where Δ = \{(1, ..., k, ..., n) =
\{(b₁₁ ... b₁m, ..., b_km, ..., b_n₁ ... b_n_m)} and b_k₁ \neq 0”

meaning that the attaching point k of x is synthesized from
the attaching point b_k₁ of x₁, as shown in Fig. 16.

As an example, some sets calculated by the function GFIRST for the positional grammars of Examples 5.1 and 5.2 follow:

\[
\begin{align*}
\text{GFIRST}(DFL, 1) &= \{(\text{arg}, 1), (\text{op}, 3), (\text{mux}, 4)\} \\
\text{GFIRST}(IF, 1) &= \{(\text{test}, 3)\} \\
\text{GFIRST}(C, 1) &= \{(\text{begin}, 1)\} \\
\text{GFIRST}(S, 1) &= \{(\text{function}, 1), (\text{predicate}, 1)\} \\
\text{GFIRST}(R, 1) &= \{(\text{end}, 1)\}
\end{align*}
\]

Function \text{Positional\_Query}(\text{NEXT})

\[
\begin{align*}
\text{\text{NEXT} is an entry of the column next} \\
\text{case NEXT of} \\
\text{NEXT = SP:} \\
\quad \text{return the row index in D_p to the first object to parse} \\
\text{NEXT = ANY:} \\
\quad \text{if all the objects have been marked and COUNTER(r) = 0 for each r then} \\
\quad \text{return the row index 0 in D_p pointing to the end-of-input symbol $} \\
\quad \text{else return “syntax error”; exit} \\
\text{NEXT = (REL', x), where REL acts on the syntactic attribute} \\
\text{k of x} \\
\quad \text{begin} \\
\quad \text{let z be the (i + 1)th object below the stack top} \\
\quad \text{if there exists in D_p exactly one nonmarked object b} \\
\quad \text{with syntactic attribute j such that (b, j) is in} \\
\quad \text{GFIRST(x, k), z REL b holds and the relation REL} \\
\quad \text{acts on a syntactic attribute of z and the syntactic} \\
\quad \text{attribute j of b, respectively then begin} \\
\quad \quad \text{if REL is an explicit relation then} \\
\quad \quad \quad \text{decrease by 1 the entry in the array} \\
\quad \quad \quad \text{COUNTER corresponding to the} \\
\quad \quad \quad \text{explicit relation z REL b} \\
\quad \quad \quad \text{return row index of b in D_p} \\
\quad \quad \text{else return “syntax error”; exit} \\
\quad \text{end} \\
\text{endcase}
\end{align*}
\]

For the case of the PG\_Class Flow Graph, the relation REL' in the entry NEXT corresponds to a relation h_i \_ k with z h_j b holding between the (i + 1)th object z below the stack top and exactly one nonmarked object b in D_p with attaching point j such that (b, j) is in GFIRST(x, k). In other words, when NEXT = (h_i \_ k, x):

1) if x is a terminal NAPE, Positional\_Query returns the index in D_p of a nonmarked NAPE whose name is x and whose kth attaching point is joined to the ith attaching point of z.

2) if x is a nonterminal NAPE (see Fig. 17), Positional\_Query returns the index in D_p of an unmarked terminal NAPE b whose jth attaching point is joined to the ith attaching point of z. The couples (x, k) and (b, j) are such that b is a terminal that begins a positional sentence derived from x and the kth attaching point of x is synthesized from k by successively applying the Δ rules in the derivation.

In general, if no objects or more than one is found, then Positional\_Query returns a parsing error.

We restrict our attention to positional grammars for which whenever the function GFIRST(x, k) is invoked by the function Positional\_Query on the nonterminal x, for each production x \_ x_1 R_1 x_2 ... x_i ... R_m \_ x_m, the syntactic attribute k of x is always synthesized from one of the syntactic attributes of the leftmost grammar object x_i in the right-hand side of the production.

For positional grammars in the PG\_Class Flow Graph, this means that if the rule Δ in the production above is given by:

\[
\Delta = \{(1, ..., k, ..., n) = \{(b₁₁ ... b₁m, ..., b_km, ..., b_n₁ ... b_n_m)\}
\]

then the condition b_k₁ \neq 0 is always true.

This does not represent a limitation, since it is always possible to reduce a positional grammar to this case by reorganizing its productions.
The function \textit{Test} shown below verifies that the grammar object to be pushed on the stack top is properly related to a grammar object already in the stack.

**Function Test(COND)**

\[
\begin{align*}
\text{COND} &= (REL, x) \text{ where } x \text{ is either a terminal or a nonterminal} \\
\text{let } z &= \text{the } (i + 1)\text{th object below the stack top} \\
\text{if } z &= \text{REL} \text{ x holds} \\
\text{then begin} \\
\text{if } REL &= \text{is an explicit relation then} \\
&\text{decrease by 1 the entry in the array COUNTER} \\
&\text{corresponding to the explicit relation } z \text{REL} \text{ b} \\
&\text{return true} \\
\text{end else return false} \\
\end{align*}
\]

The pLR parsing program is outlined in Algorithm 6.1.

**Algorithm 6.1.** The pLR parsing program.

**Input:** A picture \( p \), stored in a dictionary \( D_p \) and a pLR parsing table.

**Output:** A bottom-up parse of \( p \), if \( p \) is syntactically correct, otherwise error.

**Method:** At the beginning, the initial state \( s_0 \) is on the top of the stack.

\textbf{repeat forever} \\
begin \\
\text{let } s &= \text{the state on the top of the stack} \\
\text{if } next[s] &= \text{is defined then} \\
&\text{set } ip &= \text{Positional\_Query(next[s])} \text{ and} \\
&\text{mark the corresponding entry in } D_p \\
\text{let } b &= \text{the grammar object pointed by } ip \\
\text{if } action &= [s, b] = “\{REL_h^1, \ldots, REL_h^n\}” \\
&\text{shift } s’ &= \text{then} \\
\text{if } n &= 0 \text{ or Test(REL_h^1, b) is true for each } 1 \leq i \leq n \\
&\text{then} \\
&\text{push } b &= \text{then } s’ \text{ on the top of the stack} \\
\text{else “syntax error”} \\
\text{else if } action &= [s, b] = \text{reduce } A \rightarrow x_1 R_1 x_2 \ldots R_{m-1} x_m, \Delta \\
&\text{then begin} \\
&\text{calculate the syntactic attributes of the} \\
&\text{grammar object } A \text{ from the syntactic} \\
&\text{attributes of the grammar objects} \\
&x_1, x_2, \ldots, x_m \text{ below the stack top} \\
&\text{according to } \Delta \\
&\text{pop } 2m \text{ elements out of the stack} \\
&\text{let } s’ &= \text{the state now on the top of the stack} \\
&\text{let goto(s’, A) } &= \text{“\{REL_h^1, \ldots, REL_h^n\} : s’”} \\
&\text{if } n &= 0 \text{ or Test(REL_h^1, A) is true for each} \\
&\text{1} \leq i \leq n \\
&\text{then} \\
&\text{begin} \\
&\text{push the grammar object } A \text{ then } s’ \text{ on} \\
&\text{the top of the stack output the} \\
&\text{production } A \rightarrow x_1 R_1 x_2 \ldots R_{m-1} x_m, \Delta \\
&\text{end else “syntax error”} \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\
end \\textbf{The main differences, with respect to the LR parsing program, are given by the use of the function Positional\_Query to select the next object to be parsed and the use of the function Test to verify the existence of relations between objects that have already been processed and the next object. At each step, the pLR parsing program checks the entry \text{next}[s] of the parsing table corresponding to the state } s \text{ on the top of the stack. If this entry is defined, then the pointer } ip, \text{ used to scan the input, is updated to point to the next terminal } b. \text{In this case, either } b \text{ is shifted on the stack top (action}[s, b] = “R: shift s’”), or the input picture is accepted (action}[s, b] = “acc”) \text{with } b \text{ being the end-of-input marker \$}. \text{Wheneven a shift action is required and the action condition } R \text{ is not empty, then each relation } REL_h^i \text{ in } R \text{ is tested between the } (h + 1)\text{th object below the stack top and the object } b \text{ to be shifted.} \\

If \text{next}[s] \text{is not defined, then a reduce action is required. In this case, the pointer } ip \text{ is not updated and it points to the last terminal } b \text{ shifted on the stack top. The reduction action}[s, b] = \text{“reduce } A \rightarrow x_1 R_1 x_2 \ldots R_{m-1} x_m, \Delta” \text{is accomplished by popping } 2m \text{ elements out of the stack, calculating the syntactic attributes of } A \text{ as specified by } \Delta \text{, and pushing } A \text{ on the stack top. If } s’ \text{ is the state on the stack top after popping the } 2m \text{ elements, then the next state } s” \text{of the parser is given by the entry goto(s’, A). Also in this case, the goto action may be triggered by an action condition to be verified between objects below the stack top and the object } A. \\

Fig. 18 shows the result of compiling two pictures in the flowchart visual programming environment generated by the VLCC system. The compiler uses the algorithm above for the syntax analysis of the input. In Fig. 18a, the syntax analysis is performed successfully and the flowchart is translated into its Algol-like version according to its semantics. In Fig. 18b, the parser detects a syntax error due to the presence of an extra link and the compilation process stops with an error message.

### 6.4 Parsing Time Complexity

The time complexity of the parsing algorithm depends on the number of objects in the input, on the particular PG\_class and on the grammar. Given a positional grammar \( G \), let

- \( na \) be the maximum number of attributes of a grammar object,
- \( no \) be the maximum number of grammar objects in the right-hand side of a production, and
- \( nr \) be the maximum number of relations in a tester.

Let us consider an input picture containing \( n \) graphical objects. The worst case complexity is achieved for correct input pictures when all the input objects are visited. At each step, the parser performs a shift or a reduce action. Therefore, the total number of shifts will be \( n \), while the number of reductions will be \( O(n) \). The parsing algorithm performs a shift action whenever \text{next}[s] is defined and a reduce action otherwise. Let us compute separately the time complexity for shift and reduce actions.
To perform a shift action the parsing program must first access the input and then test the action condition, if any. Let $t_q$ be the time required to perform the function $\text{Positional\_Query}$ (on next[s]). Moreover, if an action condition is to be performed, the conditioned shift depends on the number $n_r$ of relations in a tester and on the time $t_r$ to test each relation. As the push operation on the stack takes time $n_a$, the total time complexity to perform a shift action is $O(t_q + n_r \times t_r + n_a)$.

To reduce a production, the parser has to perform the following steps:

1) calculate the syntactic attributes of the left-hand side nonterminal;
2) pop the records corresponding to the right-hand side grammar objects from the stack;
3) test for conditioned goto;
4) push the record corresponding to the left-hand side nonterminal onto the stack.

Fig. 18. Two applications of the pLR parser.
The cost of step 1) depends on the particular function used to synthesize each syntactic attribute. Let \( O(t_A(n)) \) be the time required to perform this task, then the time complexity for step 1) will be \( O(na * t_A(n)) \). As the stack pop operation takes time \( na \), step 2) will cost \( O(na * na) \). Similarly to a conditioned shift, a conditioned goto (step 3) has time complexity \( O(na * tr + na) \), while the final push operation (step 4) takes time \( na \). Therefore, the total time complexity for a reduce action is \( O(na(t_A(na) + no + nr * tr)) \).

Then, the time complexity of the parser is \( O(n(na(t_A(na) + no + nr * tr) + (tq + nr * tr + na))) \) that is \( O(n(a + t + t_A)) \). For a fixed grammar, \( na, nr, and no \) are constants and the time complexity reduces to \( O(n(tq + tr + t_A)) \). The parameters \( t, tr, \) and \( t_A \) depend on the particular PG_class. For example, for the flow graph languages described by the positional grammars in the PG_class Flow Graph, the access time \( tq \) may vary from a constant \( O(n) \), depending on the chosen implementation of the input dictionary, while the test time \( tr \) is constant. Finally, the time complexity \( t_A \) for synthesizing an attaching point is constant as each tie-point is composed of at most two non-0 attaching points. Thus, for a fixed grammar in the PG_class Flow Graph the time complexity is \( O(n * tq) \). By using proper hashing techniques to implement the dictionary \( D_{lr} \), the expected time complexity reduces to \( O(n) \).

Empirical testing on a 80486DX4-100 machine shows that the response time for parsing a flowchart with 15, 120, and 250 tokens is about 0.2, 2.3, and 4.7 msec, respectively.

### 6.5 pLR Parsing Table Construction

In the following, the construction of the pLR parsing table is outlined.

A pLR(0) item of a positional grammar G is a production from G without the synthesizing rule \( \Delta \) and with a dot at some position in the right-hand side. However, a dot can never be between a relation identifier and the terminal or nonterminal on its right. Therefore, a production \( A \rightarrow X R_1 Y R_2 Z, \Delta \) yields the following four types of items:

\[
\begin{align*}
[A & \rightarrow X R_1 Y R_2 Z] \\
[A & \rightarrow X R_1 Y R_2 Z] \\
[A & \rightarrow X R_1 Y R_2 Z] \\
[A & \rightarrow X R_1 Y R_2 Z].
\end{align*}
\]

Intuitively, an item indicates how much of a production has already been examined during the parsing process and what is yet to come. For instance, the item \([C \rightarrow \text{begin}_2.1 \text{R}]\) from Example 5.2 means that the terminal \( \text{begin}_2 \) has already been seen and a nonterminal R in relation 2_I with \( \text{begin} \) is expected next.

If G is a positional grammar with starting nonterminal \( S \), then \( G' \), the augmented positional grammar for \( G \), is obtained from \( G \) by adding a new starting nonterminal \( S' \) and the production \( S' \rightarrow S \). Algorithm 6.2 for the construction of the sets of pLR(0) items for an augmented positional grammar is obtained by properly extending the corresponding one in [1].

#### Algorithm 6.2. Construction of the sets of pLR(0) items.

**Input:** An augmented positional grammar \( G' \).

**Output:** The sets of pLR(0) items.

**Method:** The items are constructed by the main routine ITEMS which calls the functions CLOSURE and GOTO

function CLOSURE(I) \( \backslash \backslash I \) is a set of pLR(0) items

begin repeat
  for each item \([A \rightarrow \alpha, R B]\) with \( \alpha \not= \varepsilon \) or \([A \rightarrow B]\) in I and each production \( B \rightarrow \gamma \) in \( G' \)
    do add \([B \rightarrow \gamma] \) to I
  until no more items can be added to I
  return I
end

function GOTO(I, x, R) \( \backslash \backslash I \) is a set of pLR(0) items; x is a grammar object; \( \backslash \backslash R \) is a sequence of tester relations

begin
  if R = \( \emptyset \) then
    let J = \{item | item = \([A \rightarrow \alpha, R B]\) with \( \alpha \not= \varepsilon \),
    such that \( [A \rightarrow \alpha, R B] \in I \}
    A [A \rightarrow \alpha, R B] \in I \}
    return CLOSURE(J)

  else if R = \( \{REL_1, \ldots, REL_n\} \) with \( n \geq 2 \) then
    let J = \{item | item = \([A \rightarrow \alpha, REL_1, REL_2, \ldots, REL_n]\) \( x, B \)
    with \( \alpha \not= \varepsilon \), such that \( [A \rightarrow \alpha, REL_1, REL_2, \ldots, REL_n] \in I \}
    return CLOSURE(J)
end

begin set C to CLOSURE \( ([S' \rightarrow S]) \)

begin repeat
  for each set of items I in C and each grammar object x
  such that GOTO(I, x, \( \emptyset \)) is not empty and not in C do
    add GOTO(I, x, \( \emptyset \)) to C;
  for each set of items I in C, each grammar object x and
  relation R = \( \{REL_1, \ldots, REL_n\} \) with \( n \geq 2 \)
  such that \( [A \rightarrow \alpha, REL_1, REL_2, \ldots, REL_n] \in I \)
  and GOTO(I, x, R) is not in C do
    add GOTO(I, x, R) to C;
  until no more sets of items can be added to C
end

The pLR(0) sets of items of an augmented positional grammar \( G' \) are incrementally constructed by the main procedure ITEMS, starting from the initial set containing the item \([S' \rightarrow S]\). Similarly to the LR case, the sets of pLR(0) items correspond to the states of a finite automaton for viable prefixes [1] where the transitions are determined by the function GOTO.

Given a set of items I containing an item with a dot before a nonterminal B, the function CLOSURE adds to I all the items with B in the left-hand side and the dot preceding the first object of the right-hand side. This means that if the non-
terminal object B is expected next, then any object starting a positional sentential form from B is expected next.  

Once a grammar object x has been seen, the function GOTO determines the set of items containing the objects that can be seen next. This function differs from the function GOTO for LR string parsing in that it also takes a sequence of tester relations as input. Therefore, if a set of items I contains the two items \([A \rightarrow \alpha . \langle REL^{h_1}, R_1 \rangle \times \beta]\), and \([B \rightarrow \gamma . \langle REL^{h_2}, R_2 \rangle \times \delta]\) where \(R_1\) and \(R_2\) are sequences of tester relations and \(R_1 \neq R_2\), then the set GOTO(I, x, R_2) differs from the set GOTO(I, x, R_1), i.e., there are two different transitions on the grammar object x in the finite automaton for viable prefixes. 

Algorithm 6.3 for the construction of a pLR parsing table follows.

Algorithm 6.3. Construction of the pLR(1) parsing table. 

**Input:** An augmented positional grammar \(G'\).  
**Output:** The pLR(1) parsing table for \(G'\).  

**Method:**  

1) The entries for state i and nonterminals \(X\) of the pLR grammar are determined as follows:  

**SHIFT ENTRIES**  

a) If \([A \rightarrow \alpha . \langle REL^{h_1}, R_1 \rangle \times \beta]\) or \([A \rightarrow \alpha \beta]\) in I, and GOTO \((I_x, a, O) = I_1\), then insert “: shift j” into \(action\) \([i, a]\).  

b) If \([A \rightarrow \alpha . \langle REL^{h_1}, REL^{h_2}, \ldots, REL^{h_n} \rangle \times \beta]\), a \(b\), with \(n \geq 2\), is in I, and GOTO \((I_x, a, \langle REL^{h_2}, \ldots, REL^{h_n} \rangle) = I_1\), then insert “: \(REL^{h_2}, \ldots, REL^{h_n}\) : shift j” into \(action\) \([i, a]\).  

c) If \([A \rightarrow \alpha]\) is in I, then for each terminal \(a\) insert “reduce \(A \rightarrow \alpha\) into \(action\) \([i, a]\). 

**REDUCE ENTRIES**  

d) If \([A \rightarrow \alpha . \langle REL^{h_1}, REL^{h_2}, \ldots, REL^{h_n} \rangle \times \beta]\), with \(n \geq 1\), is in I, then insert (\(REL^{h_1}, x\)) in next \([i]\).  

e) If \([S' \rightarrow S]\) is in I, then insert SD into next \([i]\).  

3) The entries for state i and nonterminals X of the goto part are determined as follows:  

a) If \([A \rightarrow \alpha . \langle REL^{h_1}, REL^{h_2}, \ldots, REL^{h_n} \rangle \times \beta]\), with \(n \geq 2\), is in I, and GOTO \((I_x, X, \langle REL^{h_2}, \ldots, REL^{h_n} \rangle) = I_1\), then insert “: \(REL^{h_2}, \ldots, REL^{h_n}\) : \(j\)” into goto \([i, X]\).  

b) If \([A \rightarrow \alpha . REL^{h} \times \beta]\) or \([A \rightarrow \alpha \times \beta]\) is in I, and GOTO \((I_x, O) = I_1\), then insert “: \(j\)” into goto \([i, X]\).  

The action and goto parts of the pLR parsing table are constructed as in LR parsing tables. Action conditions and the entries in the column next are constructed as follows:

- a shift or goto action in state \(i\) has a sequence of tester relations as an action condition if and only if the set of items corresponding to state \(i\) contains an item with a dot preceding a sequence \(R_1\), containing the tester relations, and a terminal or a nonterminal, respectively.  
- the entry next[i] contains the pair \(\langle REL^{h}, x\rangle\) if and only if the set of items corresponding to state \(i\) contains an item with a dot preceding a sequence \(R_1\), with \(REL^{h}\) as its driver relation, and the grammar object \(x\).

### 6.6 pLR Parsing Table Conflicts

A conflict in a pLR parsing table arises when multiple actions are contained in a single entry in the action, goto, or next parts. Besides the usual shift-reduce and reduce-reduce conflicts, [1], a pLR parsing table may present shift-shift, goto-goto, and positional conflicts.

A shift-shift conflict occurs whenever multiple conditioned shift actions are present in a single entry of the action part. Analogously, a goto-goto conflict occurs whenever multiple conditioned goto actions appear in a single entry of the goto part. Shift-shift or goto-goto conflicts are generated whenever a set of pLR(0) items contains two or more items with the dot preceding the same grammar object but with different sequences of tester relations.

Finally, a positional conflict occurs whenever multiple values \(REL^{h}, x\) are present in a single entry of the next column. This conflict is generated whenever a set of pLR(0) items contains two or more items with the dot preceding different pairs of driver relations and grammar objects.

A positional grammar for which it is possible to construct a pLR parsing table without conflicts is said to be a pLR grammar. As an example, the positional grammar of Example 5.1 is a pLR grammar, as shown by its parsing table in Fig. 15.

### 7. Applicability of pLR Parsing

In this section we show the properties that a positional grammar must satisfy for the applicability of the pLR methodology.

To help the intuition, let us note that, if \(G\) is the string grammar obtained by deleting all the relations and the rules \(\Delta\) from the productions of a pLR grammar \(PG\), then the action and goto parts of the parsing tables for \(PG\) and \(G\) are the same, except for the action conditions that may appear in the pLR parsing table.

Moreover, during the parsing process, a pLR parser generates a sequence of calls to the function \(\text{Positional}_{\text{Query}}\) that linearizes the input at run-time. In particular, for each input picture \(\Pi\) generated by a pLR grammar \(PG = (N, T, S, P, POS, PE)\) by applying PE to a positional sentence \(\Sigma\), the function \(\text{Positional}_{\text{Query}}\) must guarantee that whenever the parser is applied to \(\Pi\) the input objects are scanned in the same order as they appear in \(\Sigma\). Note that the positional sentence \(\Sigma\) that generates \(\Pi\) is unique since PG is pLR.

Based on these considerations it is possible to prove that any picture \(\Pi\) accepted by the pLR parser of a positional...
that if PG is a pLR grammar whose parser accepted by pLR parsing algorithm.

In other words, the parser $P_{PG}$ of a pLR parsable grammar PG does never incur in run time conflicts. The following theorem gives the conditions for completeness of the pLR parsing algorithm.

**Theorem 7.2 (Completeness).** Let PG be a pLR parsable positional grammar and $P_{PG}$ its pLR parser. If $\Pi \in L(PG)$ then $\Pi$ is accepted by $P_{PG}$.

### 8 Handling Parsing Conflicts

There are two causes for a PG grammar not to be pLR parsable: parsing table conflicts and run time conflicts. In this section, we show some heuristics for handling parsing table and run time conflicts, and give an early characterization of non-pLR parsable grammars.

#### 8.1 Handling Parsing Table Conflicts

Ambiguous constructs are handled by adding heuristics. Traditional shift-reduce and reduce-reduce conflicts in a pLR parsing table are solved by using the disambiguating rules of YACC.

On the other hand, shift-shift and goto-goto conflicts are solved by ordering the conditioned actions present in the same entry. The parser tests the action conditions sequentially and executes the first action whose condition is verified.

Analogously, positional conflicts are solved by ordering the values $(REL^{p}, x)$ in the same entry of the column next. The parser invokes the function $Positional\_Query$ on each value of the sequence, until a new symbol is returned.

Similarly to YACC, in the VLCC system, the order of multiple values in the same entry of the parsing table depends on the order of the items in the same set.

Fig. 19 shows the parsing table for the positional grammar of Example 5.2. The entry $Goto[10, C]$ contains the sequence of actions "$12" and "$13." Whenever the parser needs to reduce the nonterminal C and the state resulting from the stack pop operation is state 10, then the entry $Goto[10, C]$ is checked. If the tester relation $1 \_2$ holds between the symbol on the stack top and the recognized nonterminal C, the action "goto 12" is executed, otherwise the action "goto 13" is executed. The first action corresponds to the parsing of the body of a WHILE loop construct, while the second one to the parsing of the THEN part of an IF construct.

Note that the number of action conditions in a state entry cannot be greater than the number of items in the set of items corresponding to that state. As the size of the item sets is bounded by the constant $np(no + 1)$, where $np$ is the number of productions in the grammar and $no$ is the maximum number of objects in a production, the asymptotic time complexity of the parsing algorithm does not change.
8.2 Handling Run Time Conflicts

In this section, we give some heuristics that are being implemented in the VLCC system in order to solve run-time conflicts.

As an example of a non-pLR parsable grammar, let us consider the following positional grammar from the PG_class Flow Graph for the generation of a pattern for the letter “A” (taken from [12]). In this example the relation JOINT is visualized by overlapping attaching points.

N = {A, SIDE}
T = {st}
POS = {JOINT}
P = {1) A \rightarrow SIDE 1_1 SIDE (2_1, 2_2) st
2) SIDE \rightarrow st 1_2 st

The single terminal NAPE is st which represents a straight line with two attaching points at the ends. The non terminal NAPEs for the grammar are A and SIDE. The former has no attaching points, while the latter has two attaching points. The visual productions of the grammar are depicted in Fig. 20. The first production defines an “A” as being composed of two SIDES plus a crossbar st, while the second one defines a SIDE as the connection of two st.

The rightmost derivation of the letter “A” is shown below.

A \Rightarrow [SIDE_1 1_1 SIDE_2 (2_1, 2_2) st_3]
\Rightarrow [SIDE_1 1_1 [st_3 1_2 st_4, \Delta] (2_1, 2_2) st_3]
\Rightarrow [[st_1 1_2 st_3, \Delta] 1_1 [st_3 1_2 st_4, \Delta] (2_1, 2_2) st_3]

Fig. 20. The visual productions of the “A” grammar.

The subscripts of the st terminals in the derivation indicate the scanning order of the pLR parser, as shown in Fig. 21.

Even though the parsing table does not present conflicts, a run time conflict occurs during the parsing process. To show this, let us consider the input in Fig. 21. After that st_1 and st_2 have been recognized and SIDE_1 has been reduced, the parser shifts st_3 and goes from state 2 to state 3. Here, the next terminal has to be computed by applying the function Positional_Query to next [3] = (1_2, st). The terminals retrieved are st_1 and st_5 since both are in relation 1_2 with st_3. At this point, it cannot be decided which terminal should be selected for correctly continuing the parsing process, and then, the function Positional_Query returns a syntax error.

Note that if we change the first production of the grammar into the following equivalent production:

1) A \rightarrow SIDE 2_1 st (1_1, 2_2) SIDE

the run time conflict does not occur anymore. In fact, the scanning order of the input changes as shown by the subscripts in the input representation given in Fig. 24.
Once the parser has recognized the terminals $s_1, s_2, s_3,$ and $s_4$ the only terminal returned by $\text{Positional\_Query}(1, 2, st)$ is $s_5,$ since the terminal $s_3$ has already been examined and marked.

On the other hand, there are constructs where run time conflicts can be solved without modifying the original grammar, but slightly changing the behavior of $\text{Positional\_Query}.$ Intuitively, this case occurs whenever $\text{Positional\_Query}$ detects more than one token whose scanning order is not relevant to the correct parsing process. Then, any of the detected tokens can be chosen as next input.

As an example, consider the productions in Fig. 25, graphically depicted in Fig. 26, that add the construct “CASE” to the flowchart language of Example 5.2:

\[
\begin{align*}
8) \quad S & \rightarrow \text{startcase} \ 2.1^* \ SEQ \ \langle 1.2, 1.2 \rangle \ \text{endcode} \\
9) \quad SEQ & \rightarrow \text{SEQ} \ 1.1^* \ \text{value} \ \langle 2.1, 2.1 \rangle \ C \\
10) \quad SEQ & \rightarrow \text{value} \ 2.1 \ C
\end{align*}
\]

Fig. 25. Productions for the construct CASE.

Fig. 27 shows a picture generated by the productions above where the compound statement $C$ has been left unspecified. Note that the NAPE identifying $\text{startcase}$ is connected with three NAPEs all identifying a $\text{value}.$

During the parsing process, the recognition of the NAPE $\text{startcase}$ produces the call to $\text{Positional\_Query}(2.1, SEQ)$ which detects from the input flowchart the three instances of the NAPE $\text{value}$ (as $\text{GFIRST}(\text{SEQ}, 1) = \{(\text{value}, 1)\}).$ As the syntactic correctness of the CASE construct does not depend on the order of appearance of the $<\text{value}, \text{compound statement}>$ structures, any returned NAPE $\text{value}$ can be chosen as the next object to parse. In this case, the call to $\text{Positional\_Query}(2.1, SEQ)$ should not signal a syntax error, but select a $\text{NAPE value}$ randomly and return it to the parser. The marking operation guarantees that the selected NAPE is not going to be considered again in successive queries.

In general, relations with the characteristics of the relation 2.1 above must be explicitly identified in the positional grammar by marking them with a ‘*’ (as in the productions 8 and 9 of Fig. 25). In the PG_class Flow Graph, these relations are graphically represented by a bold polyline as shown in Fig. 26.

The modification of the function $\text{Positional\_Query}$ consists of the addition of the following new case:

\[
\text{NEXT} = (REL^{i*}, x):
\]

\[
\begin{align*}
\text{begin} & \quad \text{let } z \text{ be the } (i + 1)\text{th object below the stack top} \\
\text{if} & \quad \text{there exists in } D, \text{ a nonempty set } S \text{ of nonmarked objects } b \text{ with syntactic attribute } j \text{ such that } (b, j) \text{ is in } \text{GFIRST}(x, k), \text{ then } z \text{ REL } b \text{ holds and the relation } REL \\
& \quad \text{involves a syntactic attribute of } z \text{ and the syntactic attribute } j \text{ of } b, \text{ respectively} \\
\text{then} & \quad \text{begin} \\
& \quad \text{randomly select an object } b \text{ from } S \\
& \quad \text{if } REL \text{ is an explicit relation then decrease by 1 the entry in the array COUNTER corresponding to the explicit}
\end{align*}
\]
A positional grammar PG in the PG_class Flow Graph may give rise to run time conflicts whenever one of the following two cases, schematized in Figs. 28a and 28b, respectively, holds.

Case (a). There exists a production in PG whose right-hand side contains a relation $h.k$ connecting grammar objects $x$ and $z$, and there exists a production in PG having in its right-hand side a driver relation $m.n$ connecting grammar objects $y$ and $s$ such that $(z, k) \in \text{ATTACH}(x, h)$ and $\text{GFIRST}(z, k) \cap \text{GFIRST}(s, n) \neq \emptyset$.

Case (b). There exists a production in PG whose right-hand side contains a driver relation $h.k$ connecting a grammar object $x$ to a nonterminal $Z$, and there exists a production in PG defining a nonterminal $Y$ with attaching point $m$, such that $(Y, m) \in \{(Z, k) \cup \text{ATTACH}(x, h) \mid m \in \text{ATTACH}(x, k) \}$ and $Y$ is composed of at least two grammar objects $u$ and $v$ whose attaching points $p$ and $q$, respectively, are synthesized to form the attaching point $m$ of $Y$, i.e., $(u, p) \in \text{ATTACH}(x, k)$ and $(v, q) \in \text{ATTACH}(x, k)$.

As an example, case (a) occurs for the positional grammar generating the letter “A” in Fig. 21; in fact:

- the right-hand side of production (1) contains the tester $h.k = 2.2$ connecting $\text{SIDE}_2$ to $s_5$, (i.e., $x = \text{SIDE}, z = s$).
- the right-hand side of production (2) contains a driver $m.n = 1.2$ connecting the two terminals $s_3$ and $s_4$ (i.e., $y = s = st$).
- $(s_3, 1) \in \text{ATTACH}(<\text{SIDE}_2, 2>)$.
- $\text{GFIRST}(z, k) \cap \text{GFIRST}(s, n) = \text{GFIRST}(s, 2) \cap \text{GFIRST}(s, 2) = (st, 2)$.

It is easy to verify that case (b) occurs for the productions for the construct CASE in Fig. 25.
9 Conclusion and Further Research

We have presented an efficient LR-like parsing methodology for visual languages based upon the positional grammar model. This methodology is currently used within the VLCC system for the automatic generation of visual language compilers [7].

At present, we have used the pLR methodology within the VLCC system on several types of visual languages. The most meaningful languages implemented using VLCC include flowchart languages annotated with Pascal-like code, two-dimensional arithmetical expressions, structured and semi structured flowcharts, document layouts, chemical structures, logic diagrams, data-flow graphs, electrical circuits, and finite automata graphical layouts. We are currently testing the applicability of our model to other classes of visual languages; in particular, the parser generation for a subset of statecharts is in progress, in order to experiment the ability of the system in managing the incremental definition of new visual languages starting from already generated ones.

Further research includes also the enhancement of the pLR parser embedded in the VLCC system for an extended class of flow graph languages presenting structure sharing [20]. An early extension of the pLR methodology for dealing with structure sharing flow graphs has been introduced in [8]. The derivation mechanism of positional grammars has been modified in order to allow the same instance of a non-terminal to be referred to in multiple points of a derivation. The pLR parsing program works accordingly to the new derivation mechanism. It exploits the work that has already been done for recognizing a structure, so avoiding to repeat the visit of the structure. As an application, the extended model used on the positional grammar of Example 5.1 allows the visit of the structure. As an application, the extended model used on the positional grammar of Example 5.1 allows the visit of the structure. As an application, the extended model used on the positional grammar of Example 5.1 allows the visit of the structure.

In order to treat more general visual languages, we are making deeper studies on the theoretical aspects of the applicability of the pLR methodology and, in particular, on a complete characterization of the run time conflicts for which early results have been given in this paper. Efforts are being made towards the definition of algorithms for the construction of a new pLR parsing table whose analysis allows the easy detection of non-pLR parsable grammars.

Appendix

Definition 1. Given a positional grammar PG and a positional sentence Σ derived by PG, let x₁, x₂, ..., xₘ be the terminals ordered as they appear in Σ, and REL₁, REL₂, ..., RELₘ₋₁ be the driver relations in Σ, where each RELᵢ is between xᵢ and xᵢ₊₁ for 1 ≤ i ≤ m − 1. We define Seq(Σ) as the sequence of pairs ((REL₁, x₂), (REL₂, x₃), ..., (RELₘ₋₁, xₘ)).

Definition 2. Let PG be a pLR grammar and PPG be its corresponding pLR parser. Given a picture Π accepted by PPG and containing the terminals y₁, y₂, ..., yₙ, we define PQ_SeqPG(Π) as the sequence ((REL₁, y₂), (REL₂, y₃), ..., (RELₙ₋₁, yₙ)), where:

- y₁, y₂, ..., yₙ is a permutation of y₁, y₂, ..., yₙ.
- y₁ is the terminal returned by Positional_Query invoked by PPG on SP.
- RELᵢ and yᵢ, 1 ≤ j ≤ n – 1, are the first argument and the result of the (j + 1)th call to Positional_Query made by PPG, executed on Π, respectively.

Theorem 1. Let PG be a pLR grammar and PPG its pLR parser. If Π is a picture accepted by PPG then Π ∈ L(PG).

Proof. Let Σ be the positional sentence obtained by applying in reverse order the sequence of productions of PG reduced during the parsing of Π. We need to prove that the application of PE to Σ generates Π.

Let xᵢ be the terminal returned by Positional_Query invoked on SP, and PQ_SeqPG(Π) = (REL₁, x₂), (REL₂, x₃), ..., (RELₙ₋₁, xₙ). For the sake of simplicity, we assume that no tester relations are in the grammar PG.

As Π has been accepted, the counter entries corresponding to explicit relations are all 0 and then no explicit relations other than those contained in the set {REL₁, REL₂, ..., RELₙ₋₁} are in Π.

To prove the assert we only need to prove that x₁ is the first terminal in Σ and PQ_SeqPG(Π) = Seq(Σ). Remember that the action and goto parts of the pLR parsing table of PG correspond to an LR parsing table and then, by construction of Σ, x₁ is the first terminal in Σ and the sequence of terminals is the same both in PQ_SeqPG(Π) and in Seq(Σ). Moreover, if s is the state causing the shift of the i-th terminal xᵢ during the pLR parsing of Π, then xᵢ will still be shifted in state s in the LR parsing of the string x₁, x₂, ..., xₘ. Let (RELᵢ₋₁, xᵢ) be the (i − 1)th pair in PQ_SeqPG(Π). Then, the entry next[s] contains the pair (RELᵢ₋₁, x) and xᵢ is in the set GFIRST of x. This means that the set of items corresponding to the state s must contain an item of the type [A → α .RELᵢ₋₁ x β], corresponding to a production in PG leading to the recognition of Π, and then used to derive Σ. As PG is a pLR grammar, this item must be unique. Therefore, as the terminal xᵢ in the (i − 1)th pair of Seq(Σ) must be seen in the set of items corresponding to state s, the relation contained in such pair must be RELᵢ₋₁. This proves that PQ_SeqPG(Π) = Seq(Σ).

If the positional grammar contains tester relations, then by construction of Σ, the parser verifies the tester relations in the same order as they appear in Σ. This means that if a tester relation appears in Σ, then it has been verified by the parser. Moreover, as the picture Π has been accepted, the counter entries corresponding to explicit relations are all 0 and then no other ex-
explicit relation test cases other than the ones contained in \( \Sigma \) are in \( \Pi \).

**Theorem 2.** Let PG be a pLR parsable positional grammar and \( P_{PG} \) its pLR parser. If \( \Pi \in L(PG) \) then \( \Pi \) is accepted by \( P_{PG} \).

**Proof.** Let \( \Sigma \) be the positional sentence such that PE(\( \Sigma \)) generates \( \Pi \). Let \( x_i \) be the first terminal in \( \Sigma \) and Seq(\( \Sigma \)) = \( \langle (REL_1^{hi_1}, x_1), (REL_2^{hi_2}, x_2), ..., (REL_{m-1}^{hi_{m-1}}, x_m) \rangle \). For the sake of simplicity, we first discuss the case where no action conditions are in the parsing table.

We need to show that the sequence of calls to Positional_Query on the entries of the column next returns the objects in \( \Pi \) in the same order as they appear in \( \Sigma \). In fact, in this case, the pLR parsing of \( \Pi \) reduces to the LR parsing of the string \( x_1, x_2, ..., x_m \), driven by the action and goto parts of the pLR parsing table. Then, the completeness of pLR parsing is guaranteed by the completeness of LR parsing.

The parser starts from state \( I_0 \) and invokes Positional_Query on next[s] to get \( x_i \). Suppose now that the parser has processed the terminals \( x_1, x_2, ..., x_i-1 \) and has reached a state \( s \) where a shift action is required. For construction of the action and goto parts of the pLR parsing table and considering that \( \Sigma \) is derived from PG, a shift on \( x_i \) in \( s \) must be defined. Moreover, state \( s \) must correspond to the set of items containing a kernel item (11) either of the form \( [X \to \alpha \cdot REL_1^{hi_1}, x_1 \beta] \) or of the form \( [X \to \alpha \cdot REL_1^{hi_1} \cdot Z \beta] \) and \( x_i \) is in the proper GFIRST set of \( Z \). As a consequence, next[s] is defined and contains either the pair \( (REL_1^{hi_1}, x_i) \) or the pair \( (REL_1^{hi_1}, Z) \).

At this point of the parsing process, the invocation of Positional_Query on next[s] detects and returns the only terminal \( x_i \) since \( x_i \) satisfies \( REL_1^{hi_1} \) in this context and PG is a pLR parsable positional grammar, by hypothesis.

When the terminal \( x_i \) has been processed, the parser reaches state \( s \) containing the entry \( action[s, S] = \text{"accept".} \) This state corresponds to the set of items containing the item \( [S' \to S] \) and thus the entry next[s] contains ANY. As all terminals have been visited and then marked, and all the explicit relations have been examined, the Positional_Query on ANY will return \$\$, and the parser accepts \( \Pi \).

Analogously to what has been done with the driver relations, it is easy to prove that testers are executed in the same order and in the same context as they appear in \( \Sigma \).

**References**


Gennaro Costagliola received the Laurea degree in computer science from the University of Salerno, Italy, in 1987 and the MSc degree in computer science from the University of Pittsburgh, Pennsylvania, in 1991. From 1991 to 1993 he was a postgraduate visiting student at the Department of Computer Science of the University of Pittsburgh. Since 1993 he is an assistant professor at the Department of Computer Science, University of Salerno. His research interests include visual language compilers, visual programming, visual language theory, image database indexing, and picture matching on parallel architectures. He is a member of the Association for Computing Machinery, the IEEE, and the IEEE Computer Society.

Andrea De Lucia received the Laurea degree in computer science from the University of Salerno, Italy, in 1991, the MSc degree in computer science from the University of Durham, U.K., in 1995, and the PhD degree in electronic engineering and computer science from the University of Naples “Federico II.” Italy, in 1996. From 1991 to 1992 he was at the Department of “Informatics ed Applicazioni” of the University of Salerno, Italy, where his research was funded by a scholarship from the Italian National Research Council (CNR). From 1992 to 1996 he was at the Department of “Informatics ed Sistemistica” of the University of Naples “Federico II,” Italy. From 1994 to 1995 he was also a visiting postgraduate student at the Centre for Software Maintenance of the University of Durham, U.K. Since 1996 he is an assistant professor in computer science at the Department of “Ingegneria dell’Informazione ed Ingegneria Elettrica” of the University of Salerno, faculty of engineering at Benevento, Italy. His research interests include reverse engineering, reengineering, program comprehension, and visual languages.

Sergio Orefice received the Laurea degree in computer science from the University of Salerno and a PhD degree in applied mathematics and computer science from the University of Naples. He is a research associate in the Department of Applied and Pure Mathematics at the University of L’Aquila, Italy. His research interests include logic programming, visual languages, and parsing technologies for programming languages.

Genoveffa Tortora received the Laurea degree in computer science in 1978 from the University of Salerno, Italy. Since 1990 she is professor in computer science at the University of Salerno. Her research interests include software development environments, visual languages, and pictorial information systems. She is author or coauthor of more than 50 papers published in scientific journals or proceedings of refereed conferences and is coeditor of two books. She is an editor of the International Journal of Software Engineering and Knowledge Engineering. She chaired the Fourth International Conference on Software Engineering and Knowledge Engineering and the 1997 IEEE Symposium on Visual Languages. Professor Tortora is a senior member of the IEEE.