

Exploiting XPG for Visual Languages Definition, Analysis and Development

G. Costagliola,¹ V. Deufemia,² F. Ferrucci,³ C. Gravino⁴

*Dipartimento Matematica e Informatica
Università di Salerno
Baronissi(SA), Italy*

Abstract

In this paper we present the approach based on the formalism of Extended Positional Grammars (XPG) for specifying, designing and implementing visual languages. Special emphasis is put on describing the benefits deriving from the use of such formalism. Indeed, it allows us to exploit the well-established theoretical background and techniques developed for string languages in the setting of visual languages. As a matter of fact, an efficient syntactic analysis of visual languages can be effectively performed by using a suitable extension of LR techniques. Moreover, syntax-direct translations can be used to verify properties of visual sentences during a semantic analysis phase. Finally, a visual environment for a target visual language can be automatically generated by exploiting a YACC-like tool based on XPG.

1 Introduction

The importance of graphical notations in human-computer interaction and human-human communication, and the decreasing cost of hardware technologies and graphics software have caused the development of a large number of visual languages in many different application fields [4,10,14]. In particular, visual modeling languages are largely used in the software engineering field since they allow designers to provide suitable models of the system at different levels of abstraction.

One of the main question in the visual languages research field is how to specify visual languages. Informally, a visual language can be seen as a set

¹ Email:gcostagliola@unisa.it

² Email:deufemia@unisa.it

³ Email:fferrucci@unisa.it

⁴ Email:gravino@unisa.it

of diagrams which represent the valid sentences for the language and each diagram is a collection of visual symbols arranged in the Cartesian plane.

During the last years formal methods are achieving increasing importance in the context of visual languages. Indeed, much effort is presently put to develop formal techniques for specifying, designing and implementing visual languages [3,10,15,16,20]. Several of such methods are grammar-based, even if, other different approaches have been investigated in the last years, such as logic-based [12] and algebraic approaches [22]. The literature offers several grammatical formalisms for the specification of visual languages, which differ one from another under several aspects [10,15,16,20].

In this paper we focus our attention on the approach of Extended Positional Grammars which represent a direct extension of context-free string grammars where more general relations other than concatenation are allowed. Indeed, the idea behind the definition of such formalism has been to exploit the well-established theoretical background and techniques developed for string languages in the field of visual languages. The paper highlights the expressive power of the formalism which turns out to be able to describe complex visual languages and at same time to stress the benefits that can be derived from providing a specification of a language in terms of the XPG model. In particular, we show how the use of context-free style productions can simplify the development of complex visual languages allowing us to effectively adopt an incremental approach. On the defined language several tasks can be easily performed such as customization and modifications as well as the maintenance and the debug. Moreover, syntax-directed translations can be carried out based on the syntactic structure, which is the output of the syntactic analysis process, in order to verify properties of visual sentences. Analogously, code and report generation can be effectively realized by suitably exploiting the syntax structure. All of these benefits turn out to be especially interesting in the setting of visual modeling languages because they are subject to continuous changes as the history of UML diagrams shows.

Starting from a syntax and semantic specification of the language, a visual environment specific for a language can be automatically generated exploiting the Visual Language Compiler-Compiler (VLCC) system, a graphical system which is able to assist a designer in the definition of a visual language and automatically generate a corresponding visual environment [6]. In recent years, special focus has been put on the implementation of systems that generate visual environments starting from formal specifications of visual languages [2,5,6,8,9,22].

The formalism of Extended Positional Grammars provides the syntactic framework for the VLCC. The most appealing feature of VLCC is that it inherits and extends to the visual field concepts and techniques of traditional compiler generation tools like YACC [13]. This is due to the characteristics of the extended positional grammar model which represents a natural extension of the context-free grammars. Indeed, a goal of the formalism is to overcome

the inefficiency of visual languages parsing algorithms by researching suitable extensions of the well-known LR technique. As a result, some versions of the formalism have been defined as new more powerful parsing algorithms have been devised. In particular, the Extended Positional Grammars formalism is based on an extension of LR parsing, named XpLR methodology [7]. The XpLR methodology consists of algorithms to encode a positional grammar into an XpLR parsing table. Then this parsing table deterministically drives a shift-reduce syntax analysis of diagrammatic sentences. As a main difference with traditional LR parsing the input access is no longer sequential but driven by the relations contained in the positional grammars. In particular, the parser retrieves the next symbol to be analyzed by launching queries on the input sentence.

The paper is organized as follows. Section 2 describes the main characteristics of the Extended Positional Grammars. Section 3 shows an LR-based methodology for the parsing of visual languages modeled through XPGs. Section 4 illustrates how the use of Extended Positional Grammars formalism allows to extend semantic analysis techniques developed for string languages in the field of visual languages. Section 5 is devoted to describe the main features of the VLCC system. Conclusions conclude the paper.

2 XPG: a Formalism to Define Visual Languages

In this section we illustrate the main characteristics of the *eXtended Positional Grammars* (XPG, for short).

In order to represent visual sentences, the XPG formalism uses an *attribute-based approach* [7], that is a sentence is conceived as a set of attributed symbols. The attributes of each symbol can be classified in physic, syntactic, and semantic attributes. The values of the syntactic attributes are determined by the relationships holding among the symbols. Thus, a sentence is specified by combining symbols with relations. As an example, a state transition diagram could be specified by providing the symbols representing nodes and edges, and the relations between them. In particular, the syntactic attribute to express the attachment relation between the borderline of nodes and the end point of edges could be an attaching region.

In its general definition an XPG is the pair (G, PE) , where PE is a positional evaluator, and G can be seen as a particular type of context-free string attributed grammar $(N, TUPOS, S, P)$ where:

- N is a finite non-empty set of *non-terminal* symbols;
- T is a finite non-empty set of *terminal* symbols, 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 symbol;

- P is a finite non-empty set of *productions* having the following format:

$$A \rightarrow x_1 \mathbf{R}_1 x_2 \mathbf{R}_2 \dots x_{m-1} \mathbf{R}_{m-1} x_m, \Delta, \Gamma$$

where A is a non-terminal symbol, $x_1 \mathbf{R}_1 x_2 \mathbf{R}_2 \dots x_{m-1} \mathbf{R}_{m-1} x_m$ is a linear representation with respect to POS where each x_i is a symbol in $N \cup T$ and each \mathbf{R}_j is partitioned in two sub-sequences

$$(\langle REL_1^{h_1}, \dots, REL_k^{h_k} \rangle, \langle REL_{k+1}^{h_{k+1}}, \dots, REL_n^{h_n} \rangle) \quad \text{with } 1 \leq k \leq n$$

The relation identifiers in the first sub-sequence of an \mathbf{R}_j are called *driver relations*, whereas the ones in the second sub-sequence are called *tester relations*. During syntax analysis driver relations are used to determine the next symbol to be scanned, whereas tester relations are used to check whether the last scanned symbol (terminal or non-terminal) is properly related to previously scanned symbols.

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 $(N_j, Cond_j, \Delta_j)_{j=1, \dots, t}$, $t \geq 0$, used to dynamically insert new terminal symbols in the input visual sentence during the parsing process. In particular,

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

Moreover, a property that guarantee the convergence of parsing algorithms, based on XPGs, is: “for each production $A \rightarrow x_1 \dots x_m, \Delta, \Gamma$ the number of triples in Γ whose conditions can simultaneously evaluate to true must be less than m-1”. This means that no more than m-2 symbols can be inserted in the input during the application of a production.

Informally, a Positional Evaluator PE is a materialization function which transforms a linear representation into the corresponding visual sentence in the attribute-based representation and/or graphical 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, γ, A, η such that

- (i) $A \rightarrow \eta, \Delta, \Gamma$ is a production in P,
- (ii) $\beta = \delta \eta \gamma$,
- (iii) $\alpha = \delta A' \pi \gamma$, where A' is a symbol whose attributes are set according to the rule Δ and π results from the application of the rule Γ .

We also write $\alpha \stackrel{i}{\Leftarrow} \beta$ to indicate that the reduction has been achieved by applying production i . Moreover, we write $\alpha \stackrel{*}{\Leftarrow} \beta$ 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 $S \stackrel{*}{\Leftarrow} \beta$
- a *positional sentence* from S is a string β containing no non-terminals and such that $S \stackrel{*}{\Leftarrow} \beta$
- a *visual sentential form* (*visual 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 visual sentences from the starting symbol S of XPG.

In order to exemplify the above concepts let us illustrate an XPG for generating *State Transition Diagrams*.

Example 2.1 Let $STD = ((N, T \cup POS, S, P), PE)$ be the XPG for State Transition Diagrams, characterized as follows. The set of non-terminals is given by $N = \{StateTD, Graph, Node\}$ where each symbol has one attaching region as syntactic attribute, and StateTD is the starting symbol, i.e. $S = StateTD$.

The set of terminals is given by $T = \{NODEI, NODEIF, NODEF, NODEG, EDGE, PLACEHOLD\}$. The terminal symbols NODEI, NODEIF, NODEF, NODEG have one attaching region as syntactic attribute. They represent, the *initial*, the *initial and final*, the *final*, and the *generic* node, respectively, of a state transition diagram. The terminal symbol EDGE has two attaching points as syntactic attributes corresponding to the start and end points of the edge. Finally, PLACEHOLD is a fictitious terminal symbol to be dynamically inserted in the input sentence during the parsing process. It has one attaching region as syntactic attribute. The terminal symbols are graphically depicted in Fig. 1. Here, each attaching region is represented by a bold line and is identified by the number 1, whereas the two attaching regions of EDGE are represented by bullets and are identified each by a number. In the following, the notation Sym_i denotes the attaching region i of the symbol Sym . The set of relations is given by $POS = \{\mathbf{LINK}_{i,j}, \mathbf{any}\}$, where the relation

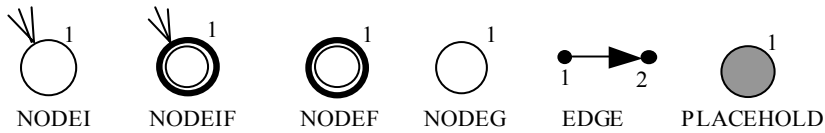


Fig. 1. The terminals for the grammar STD.

identifier *any* denotes a relation that is always satisfied between any pair of symbols. 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 State Transition Diagrams.

- (1) StateTD \rightarrow Graph
- (2) Graph \rightarrow NODEI
 Δ : (Graph₁ = NODEI₁)
- (3) Graph \rightarrow NODEIF
 Δ : (Graph₁ = NODEIF₁)
- (4) Graph \rightarrow Graph' $\langle\langle\mathbf{1_1}\rangle, \overline{\langle\mathbf{1_2}\rangle}\rangle$ EDGE **2_1** Node
 Δ : (Graph₁ = Graph'₁ - EDGE₁)
 Γ : {(PLACEHOLD; |Node₁| > 1; PLACEHOLD₁ = Node₁ - EDGE₂)}
- (5) Graph \rightarrow Graph' $\langle\langle\mathbf{1_1}\rangle, \langle\mathbf{1_2}\rangle\rangle$ EDGE
 Δ : (Graph₁ = (Graph'₁ - EDGE₁) - EDGE₂)
- (6) Graph \rightarrow Graph' $\langle\langle\mathbf{1_2}\rangle, \overline{\langle\mathbf{1_1}\rangle}\rangle$ EDGE **1_1** Node
 Δ : (Graph₁ = Graph'₁ - EDGE₂)
 Γ : {(PLACEHOLD; |Node₁| > 1; PLACEHOLD₁ = Node₁ - EDGE₁)}
- (7) Graph \rightarrow Graph' $\langle\mathbf{any}\rangle$ PLACEHOLD
 Δ : (Graph₁ = PLACEHOLD₁)
- (8) Node \rightarrow NODEG
 Δ : (Node₁ = NODEG₁)
- (9) Node \rightarrow NODEF
 Δ : (Node₁ = NODEF₁)
- (10) Node \rightarrow PLACEHOLD
 Δ : (Node₁ = PLACEHOLD₁)

Notice that Graph₁ = Graph'₁ - EDGE₁ indicates set difference and is to be interpreted as follows: “the attaching area 1 of Graph has 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 |Node₁| indicates the number of connections to the attaching area 1 of Node.

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.

Fig. 2(a-i) show the steps to reduce a state transition diagram through the extended positional grammar STD shown above. In particular, dashed ovals indicate the handles to be reduced, and their labels indicate the productions to

be used. The reduction process starts by applying production 2 to the initial state transition diagram. This causes the terminal NODE1 representing state 1 to be reduced to the non-terminal Graph. Due to the Δ rule of production 2, Graph inherits all the connections of NODE1. Similarly, the application of production 8 replaces the unique NODOG of Fig. 2(a) with the non-terminal Node. Fig. 2(b) shows the resulting visual sentential form, and highlights the handle for the application of production 4. The symbols 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 $|\text{Node}| = 4 > 1$, a new node PLACEHOLD is inserted in the input, and it is connected to all the remaining edges attached to the old Node. Fig. 2(c) shows the resulting visual sentential form.

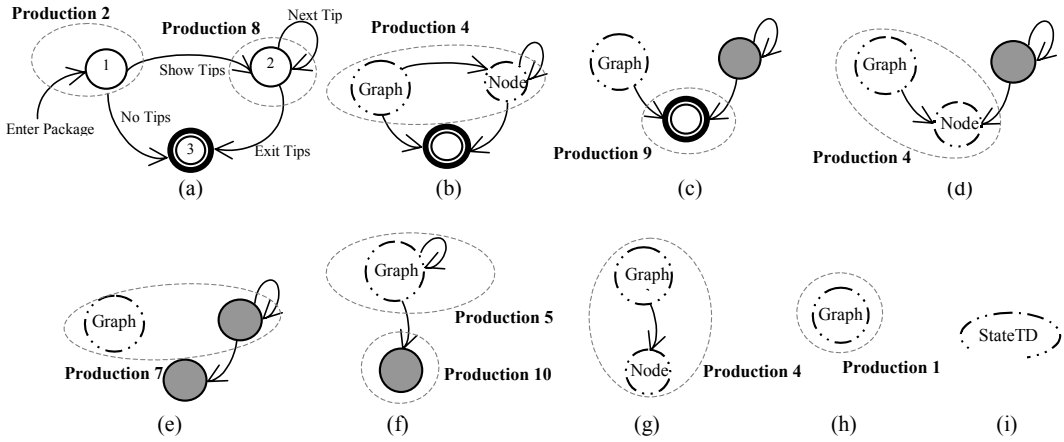


Fig. 2. The reduction process for a state transition diagram.

After the application of productions 9 and 4 the visual sentential form reduces to the one shown in Fig. 2(e). Then, production 7 reduces the non-terminals 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 Fig. 2(f)). The subsequent application of productions 10, 5, 4 and 1 reduces the original state transition diagram to the starting symbol in Fig. 2(i), confirming that the visual sentence associated to the initial state transition diagram belongs to the visual language $L(\text{STD})$.

3 Syntax Analysis of Visual Languages

The idea behind the definition of the XPG formalism is to overcome the inefficiency of the visual languages syntactic analysis by researching efficient parsing algorithms based on suitable extensions of the well-known LR technique. As a result, some versions of the formalism have been defined as new more powerful parsing algorithms have been devised, thus allowing the specification and the analysis of more complex classes of visual languages. In particular, the XPG

is based on an extension of LR parsing, named *XpLR methodology* [7], which is a framework for implementing visual systems. An XpLR parser scans the input in a non-sequential way, driven by the relations used in the grammar.

The components of an XpLR parser are shown in Fig. 3 and are detailed in the following. The input to the parser is a dictionary storing the relation-

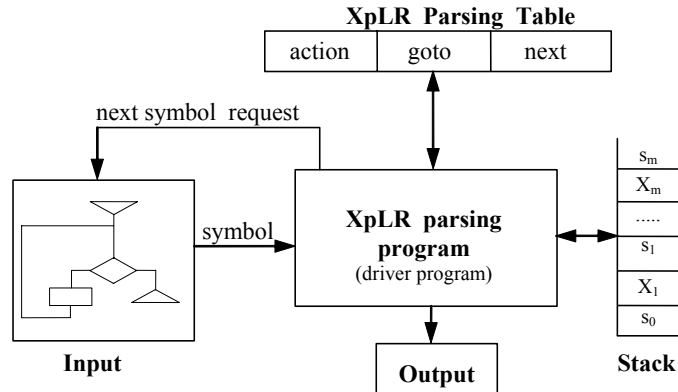


Fig. 3. The architecture of an XpLR parser.

based representation of a picture as produced by the visual editor. No parsing order is defined on the symbols in the dictionary. The parser retrieves the symbols in the dictionary by a find operation driven by the relations in the grammar.

An instance of the stack has the general format $s_0X_1s_1X_2s_2 \dots X_ms_m$, where s_m is the stack top; X_i is a grammar symbol, 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 symbol currently under examination to access a specific entry of the parsing table in order to decide the next action to execute.

An XpLR parsing table (see Fig. 4) 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 symbol to be processed. An entry $\text{next}[k]$ for a state s_k contains the pair (R_{driver}, x) , which drives the parser in selecting the next symbol to be parsed (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, Fig. 4 shows the XpLR(0) parsing table for the XPG grammar given in example 2.1. If the current state corresponds to sub-row 4.1 and the current symbol is EDGE, then if the relation 1.2 does not hold between EDGE and the first symbol below the stack top then the parser executes the conditioned action $(\overline{1.2} : \text{sh}5)$, and it goes to state 5. Otherwise

if the relation 1_2 holds between EDGE and the first symbol below the stack top then the parser executes the conditioned action (1_2 : sh6), and it goes to state 6. If it reaches state 5, due to $\text{next}[5] = (2_1, \text{Node})$, it looks for a terminal derivable from Node which is in relation 2_1 with the just seen EDGE. It can be noted that, in this case, the parser is matching the input against production (4).

The XpLR methodology resolves the shift-reduce and reduce-reduce conflicts inherited from the pLR methodology [6] by splitting a state in one or more ordered sub-states. As an example, let us consider again the XpLR(0) parsing table in Fig. 4. 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 and End-Of-Input (EOI) (see state 4.4 and production 1).

In general, an XpLR parsing table may present 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 [7].

St.	Action							Goto			NEXT
	NODEI	NODEIF	NODEF	NODEG	EDGE	PLACEHOLD	EOI	StateTD	Graph	Node	
0	:sh2	:sh3						:1	:4		(start, StateTD)
1							acc				(end, EOI)
2	r2	r2	r2	r2	r2	r2	r2				-
3	r3	r3	r3	r3	r3	r3	r3				-
4	1				$\overline{I_2}$: sh5 I_2: sh6						(I_1, EDGE)
	2				$\overline{I_1}$: sh7						(I_2, EDGE)
	3					:sh8					(any, PLACEHOLD)
	4	r1	r1	r1	r1	r1	r1				-
5			:sh11	:sh10		:sh12			:9		(2_I, Node)
6	r5	r5	r5	r5	r5	r5	r5				-
7			:sh11	:sh10		:sh12			:13		(I_I, 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				-

Fig. 4. An XpLR(0) parsing table.

As described in the previous section an XpLR parser may not converge in the analysis of a visual sentence, since the parser may get into a loop while reducing productions where the number of symbols introduced with Γ is greater or equal to the number minus one of symbols popped from the stack. Thus, the time complexity analysis of the XpLR parser is restricted to the

class of convergent parsers.

The complexity of the parser is given by the cost of the shift and reduce actions. In particular, the cost of a shift operation depends on the time to find the next symbol and the time to test the possible tester relations. While the cost of a reduce operation depends on the time to synthesize the syntactic attributes, and the time to apply the Γ rule. These values depend on the particular class of visual languages. Taking as an example the graph languages, the time complexity to parse a visual sentence containing n symbols, and with nt symbols inserted during the parsing is $O(n(n + nt))$. Indeed, the parsing time complexity on the grammar STD is $O(n^2)$ since the number of symbols introduced during the parsing is limited by the number of edges in the sentence.

4 Semantic Analysis of Visual Languages

The development of visual languages can benefit from the use of grammatical formalisms since techniques developed for string languages can be inherited in the field of visual languages. As an example, properties of visual languages could be verified during a static semantic analysis which could be carried out by exploiting the syntax structure given in output by the parsing algorithm.

To this aim, semantic attributes and semantic rules are to be added to the symbols and to the productions of the XPG obtaining a syntax-directed translation. Analogously to string languages, syntax-directed definitions can be used to specify the construction of a syntax structure summarizing the information of the input visual sentence.

In the following we will show how a semantic analysis phase performs the static verification of visual modeling languages. In particular, we will consider Statecharts [11] and UML Collaboration Diagrams [19].

In the software engineering field a static verification phase is especially important because it allows to detect possible errors during specification or design phase of a system. Statecharts are widely used to specify reactive systems. For such model it would be useful to verify the presence of loop transitions, conflicting transitions, etc. As an example, the statechart of Fig. 5 exhibits an anomalous behavior whenever the system is in *CD Paused* state and a *Pause* event happened. Indeed two different transitions are triggered by such event, causing the system to enter into both *CD Playing* and *CD Stopped* states, that should be mutually exclusive.

By adding semantic rules to the productions of XPG grammar provided in [21] the parser produces a syntax graph as output of the syntax analysis. Fig. 6 shows the syntax graph corresponding to the statechart in Fig. 5.

Algorithms on graphs can be applied to such syntax graph in order to determine the presence of specific properties such as conflicting transitions and then detect possible anomalous behaviors of the system. As an example, the conflicting transition of statechart in Fig. 5 can be detected by checking if

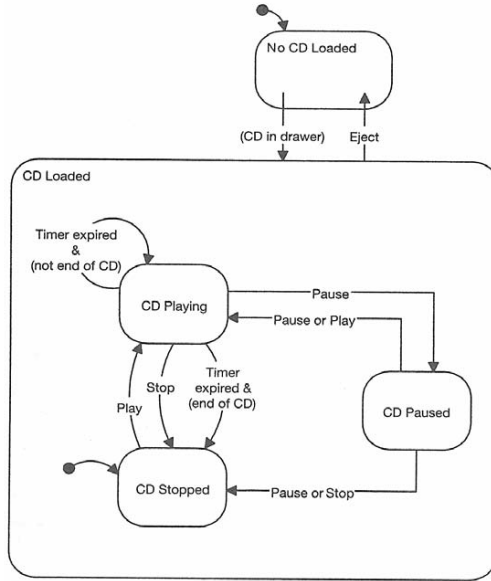


Fig. 5. A statechart that models the behavior of a CD Player user interface.

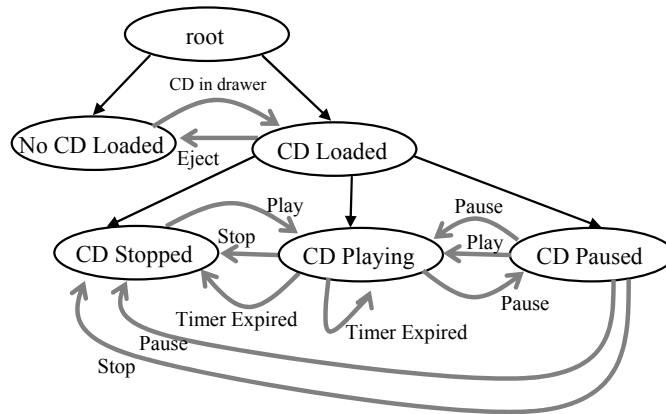


Fig. 6. The syntax graph corresponding to the statechart in Fig. 5.

there exists a node, in the syntax graph of Fig. 6, with more than one output transition with the same event (see node *CD Paused*).

Collaboration diagrams describe the set of interactions between UML classes [19]. In particular, they show the sequence of messages that pass between the linked objects. It is easy to modify the XPG grammar for state transition diagrams given in example 2.1 in order to obtain an XPG for Collaboration Diagrams.

In Fig. 7 it is depicted a collaboration diagram with an incorrect sequence of messages, as a matter of fact there are two messages with number 4. It is possible to verify such type of inconsistency by analyzing the syntax structure produced by the parser. In particular, the analysis could be effectively performed by representing the information of a collaboration diagram with a

table as shown in Fig. 8.

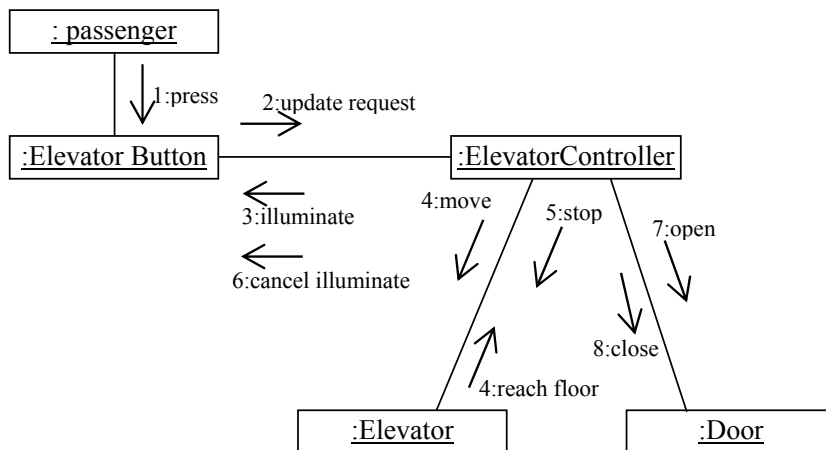


Fig. 7. A collaboration diagram for serving elevator button.

Source	Target	Message
passenger	Elevator Button	1: press
Elevator Button	ElevatorController	2: update
ElevatorController	Elevator Button	3: illuminate
ElevatorController	Elevator Button	6: cancel illuminate
ElevatorController	Elevator	4: move
Elevator	ElevatorController	4: reach floor
ElevatorController	Elevator	5: stop
ElevatorController	Door	8: close
ElevatorController	Door	7: open

Fig. 8. The syntax structure produced from collaboration in Fig. 7.

It is worth noting that it is possible to define algorithms that exploit the generated syntax structure to animate the models having a dynamic behavior. As an example, in the case of statecharts the algorithm maintains the list of active states, and the transitions from such states whose input events have occurred are performed by deactivating their source states and activating their target states.

Furthermore the syntax structure can be used to generate documents. As an example, UML sentences could be translated into the corresponding XML Model Interchange (XMI) format. XMI is a standard file format for interchanging UML designs [19].

5 An XPG-based Generator of Visual Environments

It is widely recognized that a visual language can be effectively used only if it is supported by a visual environment within which it is embedded and tightly integrated. This has motivated the research for tools which generate visual environments starting from formal specifications of visual languages [2,5,6,8,17,18,9,22].

The *Visual Language Compiler-Compiler* (VLCC) is a visual environment generation system based on the XPG model that inherits, and extends to the visual field, concepts and techniques of compiler generation tools like YACC [6,13]. Such tool assists the visual language designer in the definition of the language by assisting him/her in the specification of the symbols, the syntax and the semantics of the language, and automatically generates a visual environment starting from the supplied language specification.

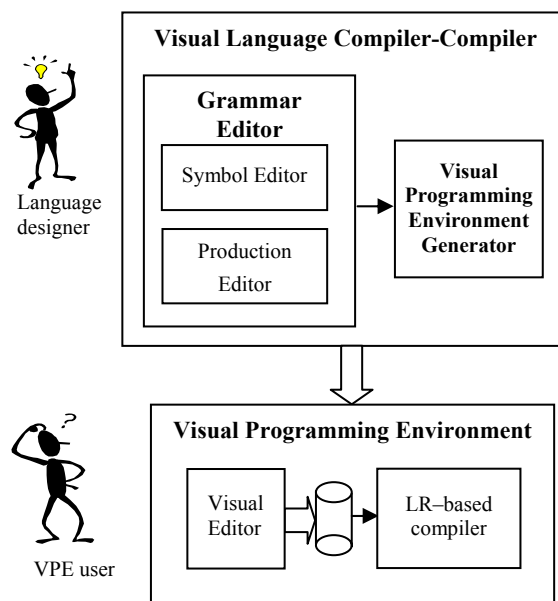


Fig. 9. The VLCC architecture.

The architecture of VLCC is shown in Fig. 9. The designer creates the terminal and the non-terminal symbols of the grammar by using the Symbol Editor. This editor works in two modes, the drawing mode and the symbol mode. In drawing mode the designer can create or modify images using the usual graphical editor facilities. In symbol mode the designer can transform an image into a grammar symbol (terminal or non-terminal) by adding the syntactic and the semantic attributes, or can modify the syntactic or semantic attributes of a symbol.

The set of terminal and non-terminal symbols are used by the language designer to create the productions using the Production Editor (see Fig. 9) that allows to define the production rules of the grammar through a text editor.

Once the XPG grammar has been defined the VPEG produces the editor and the compiler of the defined visual language.

Fig. 10 shows the visual editor window for the statecharts language together with the windows containing the terminals and the relations of the language. The user can edit a sentence of the language by selecting terminals and arranging them on the working window.

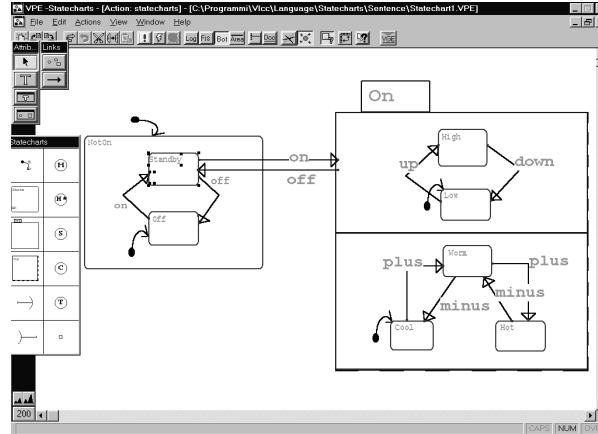


Fig. 10. The Visual Editor window for the statecharts language showing a statechart sentence that models the behavior of a blower with High and Low ventilation mode and three different temperature modes.

The availability of such tool turns out to be a useful support for a validation phase of the language development. As a matter of fact, the designer can quickly receive feedback from the customer during the language prototyping process and modify the prototype in agreement with the customer's advice. This user-centered design is especially desirable due to the nature of visual languages whose effectiveness strongly depends on the consistency between user's intention and machine interpretation.

6 Conclusions

In the paper, the XPG model has been used to illustrate the benefits that can derive from exploiting formal language descriptions for the development of visual languages. Presently grammatical formalisms for the specification of visual languages are considered of great interest. The literature offers a wide variety of such formalisms, which differ one from another under several aspects [10].

A notable feature of XPG is its ability to successfully balance the expressive power and the efficiency of the parsing algorithm. As a matter of fact, a powerful extension of the LR technique has been devised making efficient the analysis of visual languages specified using XPGs. Nevertheless, this does not reduce the expressive power of the formalism which is able to describe very complex visual languages. Indeed, the XPG is turned out to be able to capture

all the features of statecharts language that represents a very rich graphical formalism [21].

Another notable advantage of using XPG derives from the availability of an XPG-based system, which allows us to automatically obtain a visual environments starting from the syntactic and semantic specification of the language.

References

- [1] A.V. Aho, R. Sethi, and J.D. Ullman, “Compilers, principles, techniques and tools”, Addison-Wesley, New York , 1985.
- [2] R. Bardhol, “GENGED - A Generic Graphical Editor for Visual Languages Based on Algebraic Graph Grammars”, in *Procs. 1998 IEEE Symposium on Visual Languages*, Halifax, Nova Scotia, Sept 1-4, 1998, pp. 48-55.
- [3] P. Bottoni, M.F. Costabile, and P. Mussio, “Specification and dialogue control of visual interaction through visual rewriting system”, *ACM Transaction on Programming Languages and Systems*, 21(6) pp.1077-1136, 1999.
- [4] E. C. Baroth and C. Hartsough, “Experience Report: Visual Programming in the Real World”, *Visual Object Oriented Programming*, edited by M. M. Burnett, A. Goldberg & T. G. Lewis, Manning Publications, Prentice Hall, 1995, pp. 21-42.
- [5] S. S. Chok and K. Marriott, “Automatic construction of intelligent diagram editors”, in *Proceedings of the ACM Symposium on User Interface Software and Technology UIST98*, San Francisco, California, 1998, pp. 185-194.
- [6] G. Costagliola, A. De Lucia, S. Orefice, G. Tortora, “A Parsing Methodology for the Implementation of Visual Systems”, *IEEE Transactions on Software Engineering*, 23(12), 1997, pp. 777-799.
- [7] G. Costagliola and G. Polese, “Extended Positional Grammars”, in *Proceedings of 2000 IEEE Symposium on Visual Languages*, Seattle, WA, USA.
- [8] C. Crimi, A. Guercio, G. Pacini, G. Tortora, and M. Tucci, “Automating Visual Language Generation”, *IEEE Transactions on Software Engineering*, 16(10), 1990, pp. 1122-1135.
- [9] F. Ferrucci, F. Napolitano, G. Tortora, M. Tucci, and G. Vitiello, “An Interpreter for Diagrammatic Languages Based on SR Grammars”, *Procs. 13th IEEE Symposium on Visual Languages*, Capri, Italy, September, 1997, pp. 296-303.
- [10] F. Ferrucci, G. Tortora, and G. Vitiello, “Visual Programming”, in *Encyclopedia of Software Engineering*, J.J. Marciniak (Ed.), John Wiley and Sons, 2001.
- [11] D. Harel, “StateCharts: A Visual Formalism for Complex Systems”, *Science of Computer Programming*, 8, 1987, pp. 231-274.

- [12] R. Helm and K. Marriott, “Declarative Specification of Visual Languages”, *Procs. of the IEEE Workshop on Visual Languages*, 98-103 (1990).
- [13] S. C. Johnson, “YACC: Yet Another Compiler Compiler”, Bell Laboratories, Murray Hills, NJ.
- [14] G. Kent, “Automated RF Test System for Digital Cellular Telephones”, *Procs of the NEPCON West '93*, Anaheim, California, 1055-1064 (1993).
- [15] K. Marriott and B. Meyer, editors. *Visual language theory*. Springer-Verlag, 1998.
- [16] M. Minas, “Diagram Editing with Hypergraph Parser Support”, in *Procs. of 13th IEEE Symposium on Visual Languages*, Capri, Italy, Sept. 1997, 226-233.
- [17] M. Minas, “Automatically Generating Environments for Dynamic Diagram Languages”, *Procs. 14th IEEE Symposium on Visual Languages*, Halifax, Nova Scotia, Sept. 1998, pp. 70-71.
- [18] M. Minas, and G. Viehstaedt, “DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams”, in *Procs. 11th IEEE International Symposium on Visual Languages*, Darmstadt, Germany, 1995, pp. 203-210.
- [19] Object Management Group:
UML specification v1.4, OMG-Document formal/01-09-67, 2001. Available from <http://www.omg.org/technology/documents/formal/uml.htm>.
- [20] J. Rekers, A. Schurr, “A Graph Based Framework for the Implementation of Visual Environments”, in *Proceedings 12th IEEE International Symposium on Visual Languages*, Boulder, Colorado, Sept. 1996, pp. 148-157.
- [21] Statecharts Modeling Contest in Symposium on Human-Centric Computing Languages and Environments (HCC'01), Stresa, (2001). <http://www2.informatik.uni-urlangen.de/VLFM01/Statecharts>.
- [22] S.M. Uskudarli, and T.B. Dinesh, “Towards a Visual Programming Environment Generator for Algebraic Specifications”, *Procs. 11th IEEE International Symposium on Visual Languages*, Darmstadt, Germany, 1995.