Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 127 (2005) 107–125
www.elsevier.com/locate/entcs

Towards Syntax-Aware Editors for Visual Languages

Gennaro Costagliola[1] Vincenzo Deufemia[2] Giuseppe Polese[3]

Dipartimento di Matematica e Informatica
Universitá di Salerno
Fisciano(SA), Italy

Abstract

Editors for visual languages should provide a user-friendly environment supporting end users in the composition of visual sentences in an effective way. Syntax-aware editors are a class of editors that prompt users into writing syntactically correct programs by exploiting information on the visual language syntax. In particular, they do not constrain users to enter only correct syntactic states in a visual sentence. They merely inform the user when visual objects are syntactically correct. This means detecting both syntax and potential semantic errors as early as possible and providing feedback on such errors in a non-intrusive way during editing. As a consequence, error handling strategies are an essential part of such editing style of visual sentences.
In this work, we develop a strategy for the construction of syntax-aware visual language editors by integrating incremental subsentence parsers into free-hand editors. The parser combines the LR-based techniques for parsing visual languages with the more general incremental Generalized LR parsing techniques developed for string languages. Such approach has been profitably exploited for introducing a noncorrecting error recovery strategy, and for prompting during the editing the continuation of what the user is drawing.

Keywords: Visual language parsing, error-handling, syntax-aware editing.

1 Introduction

In recent years much effort has been devoted to the development of tools assisting the designer in the specification and implementation of visual environments [2,3,5,8,9]. Visual language development tools raise several interesting

[1] Email:gcostagliola@unisa.it
[2] Email:deufemia@unisa.it
[3] Email:gpolese@unisa.it

doi:10.1016/j.entcs.2004.08.050

and difficult problems, including the definition of the syntax and semantics of a graphical language, the specification of commands for editing, analyzing, and interpreting the rendering of diagrams on the screen, etc.

Several researchers have exploited visual language grammars to model the syntax of visual notations, and compiler generators to derive tools capable of processing them. However, there are many methods for implementing visual notations that are not based on rigorous syntactic or semantic modeling. They do not use parsing techniques to analyze their input drawings, but they rather implement visual notations by constructing a dedicated visual editor enforcing a syntax-directed paradigm. This means that the tool maintains an internal semantic model of a diagram being edited, and checks the consistency of the model at every editing step. Editing actions leading to inconsistent states are rejected. Although this prevents the user from drawing syntactically or semantically incorrect diagrams, it constraints the editing process, and potentially reduces user capability to abstractly reason about the drawings s/he wants to compose. In fact, the user often produces a correct target drawing by composing and manipulating several incorrect intermediate versions of it. Moreover, syntax-directed editing is uncomfortable when the user needs to perform restructuring actions, which frequently occur when sketching design diagrams for several application domains. Consequently, the syntax-directed interaction paradigm is more suitable for beginners who are learning how to use a new visual notation. Conversely, free-hand editing allows incomplete and incorrect sketches to be drawn, postponing the diagram checking phase. The order in which a diagram is drawn is not important, providing complete freedom during diagram creation. This turns out to be more comfortable for expert programmers, since they would like to be free to manipulate graphical objects without any interference due to the syntax-directed nature of the editor, invoking the parser only when they request it. The analyzer informs the user about any errors it finds during parsing and semantic processing.

A class of editors combining the positive aspects of both the approaches above is that of syntax-aware editors. Such editors prompt users into writing syntactically correct sentences. This class of editors is a compromise between the other two classes and as such it does not prevent users from entering incorrect syntactic states in a sentence. However, it informs him/her when objects are syntactically correct. Thus, the editor should make use of semantic as well as syntactic information in an attempt to provide high assistance to the user. This means detecting both syntactic and potential semantic errors as early as possible. As a consequence, error handling strategies are an essential part of the syntax-aware editing style of visual sentences.

In this paper we propose a parsing technique for the construction of visual

language editors supporting the syntax-aware editing style. The generated editors support the editing of visual sentences in free-hand style. Moreover, the underlying parsers incrementally analyze the sentences while they are entered, providing an immediate feedback to the user by highlighting correct and incorrect subsentences, and offering additional support in the construction of the sentences. In particular, during the generation of an editor the parsing technique stores information that the editor exploits to suggest the user how to construct the visual sentence in an easier and more effective way.

The approach is based on eXtended Positional Grammars (XPGs) and on a parsing technique for recognizing visual subsentences. The former is a powerful grammar formalism for modeling a broad class of visual languages, whereas the new parsing methodology is based on the incremental and pseudo-parallel version of Tomita's parsing algorithm [11] developed for string languages, and the LR-based parsing technique introduced in [7]. In particular, we have developed an algorithm for non-deterministic incremental subsentence parsing, and combined two of them for the construction of a bidirectional subsentence parser to start the parsing process from an arbitrary input symbol. Such parser has been profitably exploited for noncorrecting error recovery of visual sentences, and for prompting the continuation of what the user is drawing.

The paper is organized as follows. Section 2 reviews the related approaches. Section 3 describes the main characteristics of the XPG grammar formalism, and shows a technique for the parsing of visual subsentences of languages modeled through XPGs. Section 4 illustrates how the parser can be used to implement noncorrecting syntax error recovery and to prompt next symbols during the editing process. Finally, conclusions and further researches are discussed in Section 5.

2 Related Work

To overcome the limitations of both the free-hand and the syntax-directed editing style several approaches have been proposed.

The intelligent diagram metaphor is a recent metaphor in which the editor parses the diagram as it is being constructed, while performing error correction and collecting geometric constraints that capture the relationships between diagram components. This metaphor is supported by the editors generated with the visual programming environment generator *Penguins* [3]. In such editors the diagrams are created in free form and in any order. During diagram manipulation objects in the diagram can be moved or resized while the constraint solver maintains the semantics by preserving the geometric constraints between the diagram components. Penguins leverages a constraint solver that is

able to maintain arbitrary linear arithmetic constraints necessary for geometric error correction and diagram manipulation. The error correction mechanism used in Penguins is based on the concept of the geometric distance between sentences. By computing the geometrically closest sentence that belongs to the language, an incorrect sentence can be automatically corrected by changing attribute values of the graphical symbols.

The issue of incorporating both editing the free-hand and the syntax-directed editing modes into one editor has also been analyzed by Köth and Minas in [8]. They propose the hypergraph grammars for the specification of visual languages, and graph transformation rules for adding syntax-directed editing to the free-hand editing mode. In particular, after each editing operation the corresponding transformation rules modify the internal hypergraph, which is then reparsed to indicate the correctness and to create a valid layout. The approach implemented into DiaGen also contains an error-recovering strategy with immediate feedback to the user [10].

In *GenGED* the syntax specification of visual models is the basis for the configuration of a visual environment for syntax-directed or free-hand editing [2]. In particular, a *syntax grammar* is used to specify the editing command (i.e., rules are defined for modification, deletion, etc. of graphical elements), whereas a *parse grammar* is used to define a parser that tries to recognize the edited diagrams. These specifications based on algebraic graph transformation allow comprehensive editing and analysis of visual sentences.

An error handling strategy for the parsing algorithm based on atomic relational grammars has been proposed by Tuovinen [14]. In particular, the error recovery techniques aim at enabling the parser to continue processing the input in spite of syntactic errors rather than correcting the errors. The error handling strategy has been implemented into the Vilpert system [13], an object-oriented framework for implementing visual languages in Java.

The approach proposed in this paper allows us to include interactivity into a free-hand editor similarly to what has been done in [3]. However it does not take into account the problem of maintaining consistency between the symbols during their manipulation. On the other hand, our error handling strategy attempts to find as many errors in the input as possible, similarly to Tuovinen's and Minas's approaches, and does not take in consideration their correction. Moreover, another important aspect of our approach, not treated by the previous quoted tools, is to assist the user in the composition of the visual sentences.

3 Visual Subsentence Recognition

In this section we illustrate the main characteristics of the *eXtended Positional Grammars* (XPG, for short), and the incremental and generalized XpLR technique for parsing visual subsentences modeled through XPGs [6].

3.1 Modeling Visual Languages with XPG

In order to represent visual sentences, the XPG formalism uses an attribute-based approach [7]. In this approach a sentence is conceived as a set of attributed symbols. 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 a node and the end point of edges can be represented by an "attaching region" on that node.

More formally, an Extended Positional Grammar is the pair (G, PE), where PE is a positional evaluator, and G is a particular type of context-free string attributed grammar (N, T∪POS, 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∩T=∅;
- POS is a finite set of *binary relation* identifiers, with POS∩N=∅ and POS∩T=∅;
- S∈N denotes the *starting symbol*;
- P is a finite non-empty set of *productions* of the following format:

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

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

$$(\langle REL_{j_1}^{h_1},\ldots,REL_{j_k}^{h_k}\rangle,\langle REL_{j_{k+1}}^{h_{k+1}},\ldots,REL_{j_n}^{h_n}\rangle) \qquad \text{with } 1\leq k\leq n$$

Each $REL_{j_i}^{h_i}$ relates syntactic attributes of x_{j+1} with syntactic attributes of x_{j-h_i}, with $0\leq h_i<$j. In the rest of the paper, we will denote REL_1^0 simply as REL_1. 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*. Driver relations are used during syntax analysis to determine the next vsymbol to be scanned, whereas tester relations are used to check whether the last scanned vsymbol (terminal or non-terminal) is properly related to previously scanned vsymbols. We refer to the driver (tester, resp.) relations of $\mathbf{R_j}$ with *driver*($\mathbf{R_j}$) (*tester*($\mathbf{R_j}$), resp.).

Δ is a set of rules used to synthesize the values of the syntactic attributes of A from those of x_1, x_2,...,x_m.
Γ is a set of triples $(N_j, Cond_j, \Delta_j)_{j=1,..,t}$, $t \geq 0$, used to dynamically insert new 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,..., x_m.

Informally, a Positional Evaluator PE is a materialization function that transforms a linear representation into the corresponding visual sentence in the attribute-based representation and/or graphical representation. The attribute-based representation of a visual sentence is a list of all the objects forming the sentence together with the values of their syntactic attributes.

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

For each relation it is possible to specify a semantically opposed relation. In particular, let REL_1 and REL_2 be two relation identifiers, if $x\ REL_1\ y$ and $y\ REL_2\ x$ hold for the same pairs of symbols x and y then REL_2 is the *inverse relation* of REL_1 and vice versa. In the following, we denote with $inv(\mathbf{R})$ the inverse relation of $\mathbf{R}$. It is worth noting that for reflexive relations it happens that $inv(\mathbf{R})=\mathbf{R}$. A *reverse grammar* with respect to a XPG G=((N, T $\cup$ POS, S, P), PE), denoted with $rev(G)$, is an XPG G$'$=((N, T $\cup$ POS$'$, S, P$'$), PE), where POS$'$=inv(POS) and P$'$ is defined as in [6]. Note that L(G) is equivalent to L(rev(G)) for each XPG grammar G.

In the following we show an example of XPG grammar modeling the data flow diagram language. Let DFD be the name of the grammar, the set of nonterminals is N = {DataFD, Node}, where each symbol has one attaching region as syntactic attribute, and DataFD is the starting symbol of DFD, i.e. S = DataFD. The set of terminals is given by T = {PROCESS, STORE, ENTITY, EDGE, PLACEHOLD}. The terminal symbols PROCESS, STORE and ENTITY have one attaching region as syntactic attribute. They represent, the *processing step* node, the *data store* (or *data source*) node, and the *externally entity* node, respectively, of a data flow 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.

In Figure 1, each attaching region is represented by a bold line and is identified by the number 1, whereas the two attaching points of EDGE are

represented by bullets and are identified each by a number.

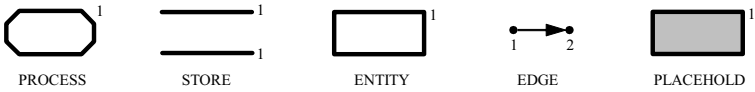

Fig. 1. The terminals for the grammar DFD.

The set of relations is given by POS = $\{\boldsymbol{LINK}_{\mathbf{h,k}}, \boldsymbol{any}\}$, where the relation identifier ***any*** denotes a relation that is always satisfied between any pair of symbols, whereas the relation identifier $\boldsymbol{LINK}_{\mathbf{i,j}}$ is defined as: "a symbol x is in relation with a symbol y iff attaching point i of x is connected to attaching point j of y", and will be denoted as **i_j** to simplify the notation. Moreover, we use the notation $\overline{\mathbf{h\_k}}$ when describing the absence of a connection between two attaching areas h and k. Notice that $inv(\mathbf{h\_k})$=**k_h**. The set of productions for DFD and $rev(DFD)$ follow.

(1) DataFD → PROCESS Δ: (DataFD_1 = PROCESS_1)	(1') DataFD → PROCESS Δ: (DataFD_1 = PROCESS_1)
(2) DataFD → DataFD' ⟨⟨***1_1***⟩,⟨$\overline{\textbf{1\_2}}$⟩⟩ EDGE ***2_1*** Node Δ: (DataFD_1 = DataFD'_1 – EDGE_1) Γ: {(PLACEHOLD; **true**; PLACEHOLD_1 = Node_1 – EDGE_2)}	(2') DataFD → Node ***1_2*** EDGE ⟨⟨***1_1***⟩,⟨$\overline{\textbf{2\_1}}$⟩⟩ DataFD' Δ: (DataFD_1 = DataFD'_1 – EDGE_1) Γ: {(PLACEHOLD; **true**; PLACEHOLD_1 = Node_1 – EDGE_2)}
(3) DataFD → DataFD' ⟨⟨***2_1***⟩,⟨$\overline{\textbf{1\_1}}$⟩⟩ EDGE ***1_1*** Node Δ: (DataFD_1 = DataFD'_1 – EDGE_2) Γ: {(PLACEHOLD; **true**; PLACEHOLD_1 = Node_1 – EDGE_1)}	(3') DataFD → Node ***1_1*** EDGE ⟨⟨***2_1***⟩,⟨$\overline{\textbf{1\_1}}$⟩⟩ DataFD' Δ: (DataFD_1 = DataFD'_1 – EDGE_2) Γ: {(PLACEHOLD; **true**; PLACEHOLD_1 = Node_1 – EDGE_1)}
(4) DataFD → DataFD' ⟨***any***⟩ PLACEHOLD Δ: (DataFD_1 = $\text{DataFD'}_1 \cup \text{PLACEHOLD}_1$)	(4') DataFD → PLACEHOLD ⟨***any***⟩ DataFD' Δ: (DataFD_1 = $\text{PLACEHOLD}_1 \cup \text{DataFD'}_1$)
(5) Node → Node' ⟨⟨***1_1***⟩,⟨$\overline{\textbf{1\_2}}$⟩⟩ EDGE Δ: (Node_1 = Node'_1 – EDGE_2)	(5') Node → EDGE ⟨⟨***1_1***⟩,⟨$\overline{\textbf{2\_1}}$⟩⟩ Node' Δ: (Node_1 = Node'_1 – EDGE_2)
(6) Node → Node' ⟨⟨***1_2***⟩,⟨$\overline{\textbf{1\_1}}$⟩⟩ EDGE Δ: (Node_1 = Node'_1 – EDGE_1)	(6') Node → EDGE ⟨⟨***2_1***⟩,⟨$\overline{\textbf{1\_1}}$⟩⟩ Node' Δ: (Node_1 = Node'_1 – EDGE_1)
(7) Node → STORE Δ: (Node_1 = STORE_1)	(7') Node → STORE Δ: (Node_1 = STORE_1)
(8) Node → PROCESS Δ: (Node_1 = PROCESS_1)	(8') Node → PROCESS Δ: (Node_1 = PROCESS_1)
(9) Node → ENTITY Δ: (Node_1 = ENTITY_1)	(9') Node → ENTITY Δ: (Node_1 = ENTITY_1)
(10) Node → PLACEHOLD Δ: (Node_1 = PLACEHOLD_1)	(10') Node → PLACEHOLD Δ: (Node_1 = PLACEHOLD_1)

Notice that $Node_1 = Node'_1 - EDGE_1$ indicates set difference and is to be interpreted as follows: "the attaching area 1 of $Node$ has to be connected to whatever is attached to the attaching area 1 of $Node'$ except for the attaching point 1 of $EDGE$". Moreover, the notation $|Node_1|$ indicates the number of connections to the attaching area 1 of $Node$. Notice that the superscripts are used to distinguish different occurrences of the same symbol. According to

these rules, a Data Flow Diagram is defined as

- a processing step node (production 1) or, recursively, as
- a DFD connected to a node through an outgoing (production 2) or incoming (production 3) edge.

A node can be either a node connected to an outgoing (production 5) or incoming (production 6) edge, or a processing step node (production 7), or a data store node (production 8), or an entity (production 9). The need for productions 4 and 10 will be clarified in the following.

Figures 2(a-i) show the steps to reduce a data flow diagram through the extended positional grammar DFD 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 1 to a processing step node. This causes the terminal PROCESS to be reduced to the non-terminal DataFD. Due to the Δ rule of production 1, DataFD inherits all the connections of PROCESS. Similarly, the application of production 8 replaces a PROCESS of Figure 2(a) with the non-terminal Node. Figure 2(b) shows the resulting visual sentential form, and highlights the handle for the application of production 2. The symbols DataFD, EDGE, and Node are then reduced to the new non-terminal DataFD. Due to the Δ rule of production 2, the new DataFD is connected to all the remaining edges attached to the old DataFD. Moreover, due to the Γ rule a new node PLACEHOLD is inserted in the input, and it is connected to all the remaining edges attached to the old Node.

After the application of productions 8, 9 and 2 the visual sentential form reduces to the one shown in Figure 2(d). Then, production 4 reduces the non-terminals DataFD and PLACEHOLD to a new non-terminal DataFD. By applying the Δ rule of production 4, the new DataFD inherits all the connections of PLACEHOLD (see Figure 2(e)). The subsequent application of productions 3, 4, 10, 2, and 4 reduces the original state transition diagram to the starting symbol in Figure 2(h), confirming that the visual sentence associated to the initial data flow diagram belongs to the visual language L(DFD).

3.2 An Incremental Subsentence Parser for Visual Languages

Parsers based on XPGs are an extension of LR parsing, named *XpLR parsing* [7]. A peculiarity of XpLR parsers is its scanning of the input in a non-sequential way (driven by the relations used in the grammar). However, this increases the occurrence of parsing conflicts. Indeed, an XpLR parser suffers from the same drawbacks as any other deterministic table-driven parser: the language grammar must be unambiguous and conform to the limitations

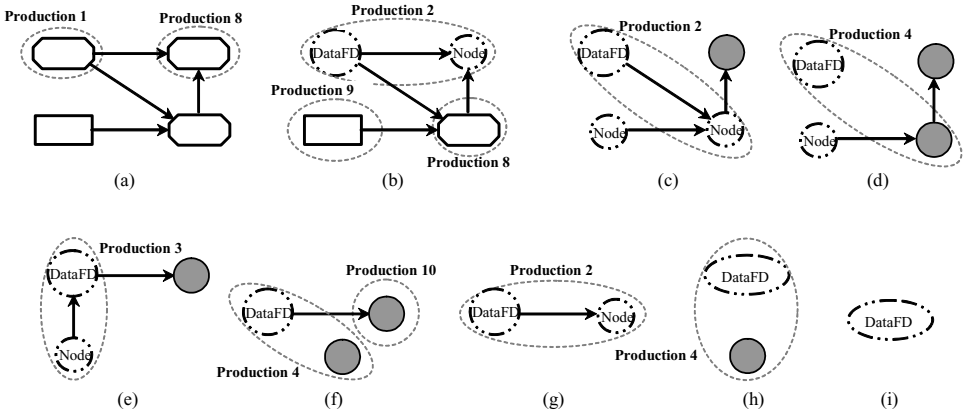

Fig. 2. The reduction process for a data flow diagram.

of the particular table-generation algorithm, which, in many cases, is quite restrictive and requires significant "grammar-hacking". Moreover, an XpLR parser as defined in [7] does not provide any feedback while the user composes a sentence. In many cases this is not desirable since a visual environment needs to be interactive in order to make the user comfortable with its use. In order to give immediate feedback to the user, a visual interactive environment requires the use of incremental parsing methods. To this aim, in the following we introduce an incremental and generalized version of the XpLR parser for recognizing visual subsentences, namely *X-Parser*, which is based on the Generalized LR parsing (GLR) [11,12].

GLR parsing is a technique for parsing arbitrary context-free grammars that utilizes conventional LR table construction methods. Unlike deterministic parsers, however, a GLR parser permits these tables to contain conflicts. The conflicts are successfully handled by using a *graph-structured stack* and by representing the possible parse tree in a compact way (the *packed shared parse forest*). Additionally, GLR permits a syntactically ambiguous grammar specification, which is necessary because the syntax of many languages, included the visual ones, falls outside the LR(k) class of languages.

The components of an X-parser are shown in Figure 3 and are detailed in the following.

The input to the incremental parser is a dictionary storing the attribute-based representation of the modified visual sentence as produced by the visual editor, a parse forest and a graph stack built on the original visual sentence. The parser matches the modified visual sentence with the yield of the parse forest, restructures the parse forest on the base of the modifications, and updates the graph stack. The match is accomplished by retrieving the objects in the dictionary through the *Fetch_Symbol* function driven by the relations

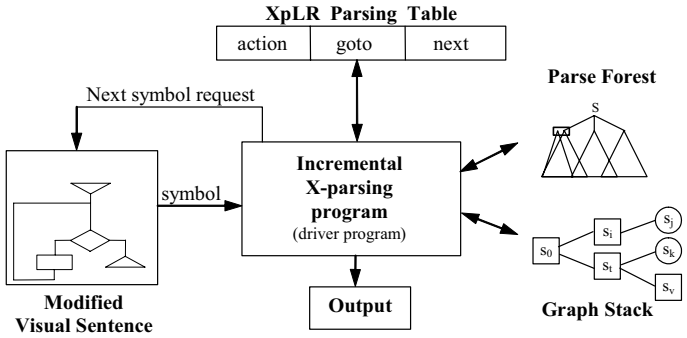

Fig. 3. The architecture of an X-parser.

in the grammar.

The graph stack has more than one stack top (usually visualized by circles). The operations of *Splitting*, *Combining* and *Local Ambiguity Packing* avoid an exponential growth of the stack during the parsing process [12]. For a highly ambiguous grammar, many parse trees might be generated for the input. The packed shared parse forest allows to share common subtrees, and to pack vertices whose parse subtrees describe the same portion of input and lead to the same state.

An XpLR parsing table (see Figure 4) is composed by a set of rows and is divided into three main sections: *action*, *goto*, and *next*. Each row corresponds to a parser state and is composed of a set of one or more sub-rows. The *action* and *goto* sections are similar to the ones used in the 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 $next[k]$ for a state s_k contains the pair $(\mathbf{R}_{\text{driver}}, x)$, which drives the parser in selecting the next symbol (derivable from x) by using the sequence of driver relations $\mathbf{R}_{\text{driver}}$.

The special entries (***start***, S) and (***end***, EOI) are used to retrieve the first symbol to be parsed and to check whether the whole input sentence has been parsed, respectively. The action and goto entries are named *conditioned actions* and have the format "$\mathbf{R}_{\text{tester}}$: *state*" and "$\mathbf{R}_{\text{tester}}$: *shift state*", respectively, where $\mathbf{R}_{\text{tester}}$ is a possibly empty sequence of tester relations. A shift or goto action is executed only if all the relations in $\mathbf{R}_{\text{tester}}$ are true, or if $\mathbf{R}_{\text{tester}}$ is empty.

The X-parser permits XpLR parsing tables to contain conflicts: when a state transition is multiply defined, the parser simply forks multiple parsers to follow each possibility. The algorithms for the construction of an XpLR parsing table are based on the notion of item [1]. An *XpLR item* of an extended positional grammar is a production without the Δ and Γ rules, and with a dot at some position of the right-hand-side. However, a dot can never be placed

between a relation identifier and the terminal or non-terminal symbol to its right [5].

St.		Action						Goto		*NEXT*
		PROCESS	STORE	ENTITY	EDGE	PLACEHOLD	EOI	DataFD	Node	
0		:sh2						:1		(***start***, DataFD)
1	1				$\overline{\textit{1\_2}}$: sh3					(***1_1***, EDGE)
	2				$\overline{\overline{\textit{1\_1}}}$: sh4					(***1_2***, EDGE)
	3					:sh5				(***any***, PLACEHOLD)
	4						acc			(***end***, EOI)
2		r1	r1	r1	r1	r1	r1			-
3		:sh11	:sh10	:sh12		:sh13			:6	(***2_1***, Node)
4		:sh11	:sh10	:sh12		:sh13			:7	(***1_1***, Node)
5		r4	r4	r4	r4	r4	r4			-
6	1				$\overline{\textit{1\_2}}$: sh8					(***1_1***, EDGE)
	2				$\overline{\overline{\textit{1\_1}}}$: sh9					(***1_2***, EDGE)
	3	r2	r2	r2	r2	r2	r2			-
7	1				$\overline{\textit{1\_2}}$: sh8					(***1_1***, EDGE)
	2				$\overline{\overline{\textit{1\_1}}}$: sh9					(***1_2***, EDGE)
	3	r3	r3	r3	r3	r3	r3			-
8		r5	r5	r5	r5	r5	r5			-
9		r6	r6	r6	r6	r6	r6			-
10		r7	r7	r7	r7	r7	r7			-
11		r8	r8	r8	r8	r8	r8			-
12		r9	r9	r9	r9	r9	r9			-
13		r10	r10	r10	r10	r10	r10			-

Fig. 4. The XpLR parsing table for DFD grammar.

Figures 5(a-e) show the application of the X-parsing algorithm during the composition of a data flow diagram. In particular, the top portion of each figure visualizes the partial sentences created during the editing, while the bottom portions show the corresponding parse shared forests. The shaded regions highlight the subtrees recovered from the previous parsing execution.

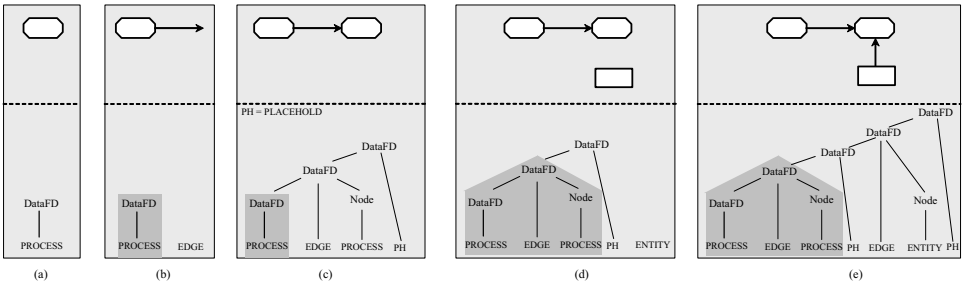

Fig. 5. Incremental parsing of a data flow diagram.

In our approach the parsing algorithm is invoked by the editor as the visual sentence is modified, and it is immediately possible to tell whether the sentence edited so far is partially or completely accepted, just by looking at the parser state. However, it is worth noting that in the LR parsing of visual languages it is difficult to establish from which symbol of a sentence

the parsing process has to start. In fact, the parsing of a DFD with an X-parser starts always looking for a PROCESS symbol since, in the grammar, it is the first reachable symbol from the starting non-terminal DataFD. This limitation prevents the parser from the possibility to recognize portions of correct sentences, and consequently prevents the editor to assist the user in the sentence composition. To this aim, the parsing algorithm has to be modified to overcome the difficulty of starting the parsing process from any symbol of a visual sentence.

In [6] it has been introduced an algorithm that allows any element of the input to be considered as the starting one and, at the same time, assures that the parsing process is not compromised. The idea is to use two parsers that proceed in parallel, scanning the input sentence in opposite directions from an arbitrary starting symbol, as shown in Figure 6.

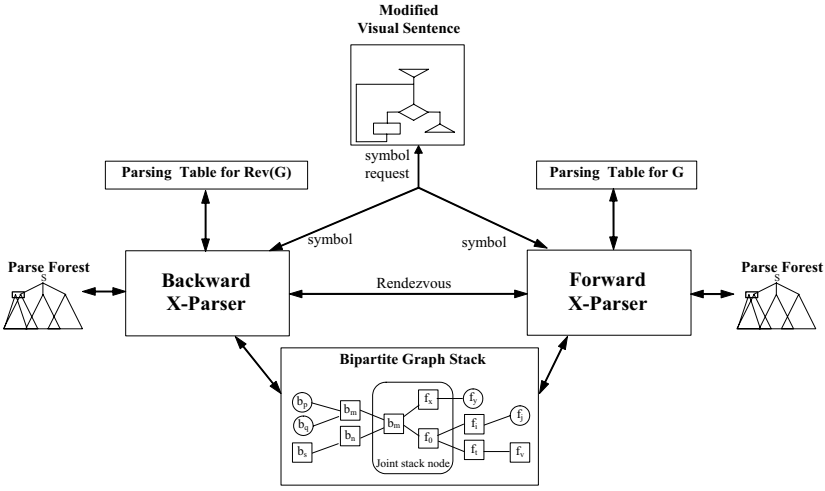

Fig. 6. The architecture of a bidirectional X-parser.

The forward and backward parser stacks can be considered as only one graph stack expanding to the right and to the left, and with two types of nodes: simple stack node and joint stack node. The latter encloses a bipartite graph whose elements are simple stack nodes from forward and backward parsers. Each stack node includes information on the state reached by the parser, the last terminal parsed, and a pointer to a node in the packed shared forest. The incremental parser must control that the rendezvous operation can be applied before reusing a subtree. Moreover, the reintroduction of the terminals is local to the couple of forward and backward parsers that execute the rendezvous.

In particular, the algorithm creates the XpLR parsing tables for the orig-

inal XPG grammar G and for its reverse version *rev*(G). For each state in G (*rev*(G), resp.) reachable after the occurrence of the starting symbol the algorithm starts an incremental X-parser, named *forward* (*backward*, resp.). The forward parsers may interact with the backward parsers only when a parser tries to reduce a production. In this case, that parser waits for a rendezvous, i.e., an opposite parser attempting to apply the reverse version of the same reduction (and the parsers have parsed different symbols).

A sentence w is recognized by the bidirectional parser if there exist a backward parser B and a forward parser F such that:

(1) each symbol of w is visited by only one of the two parsers, except the starting one that is visited by both, and,

(2) if $s_1 = xw_1$ and $s_2 = xw_2$ are the subsentences recognized by B and F, respectively, then $w = inv(w_1)\ x\ w_2$. Notice that x corresponds to the symbol from which the parsing starts.

4 Using Bidirectional X-Parsers for Syntax-Aware Editing of Visual Sentences

In this section we show how to use the bidirectional parser for implementing noncorrecting syntax error recovery and for prompting the continuation of what the user is drawing.

4.1 Noncorrecting Syntax Error Recovery

For our application of visual language parsing in a syntax-aware editor, if a visual sentence fails to satisfy the rules of the language the parser must be able to indicate the piece of input that caused the failure. Further, the parser must be able to recover from syntactic parse errors in order to allow the parsing of the remaining part of a sentence.

Notice that the parsing process of a visual sentence may fail

(i) when the *Fetch_Symbol* function does not find the requested symbol, or

(ii) when the last analyzed symbol is not properly related through the tester relations to the previously analyzed symbols.

Moreover, as defined in [14] a global parsing failure means that (1) all the parallel parsers initiated by the arbitrary starting symbol failed, or that (2) at least one of the parsers succeeded, but there is unprocessed input left. In the first case, the parser-defined error is the set of input symbols causing the parse action failures at the end of the most successful parse paths starting from the arbitrary point. Figure 5(b) shows an example of this type of failure

caused by the EDGE symbol. In the second case of failure, the parser-defined error is the set of extra input objects. Figure 5(d) shows a correct sentence with an extra symbol STORE causing this type of failure. Note that there can be several equally successful parse paths.

If the parser finds an error, it could try to correct it in order to continue parsing. However, if the parser makes false assumptions about the kind of error encountered then spurious errors are easily introduced.

In our approach, the parser does not make any assumption about how to correct the error, or skip input until a trusted symbol is found. In particular, if a parser detects a parsing failure on some symbol, the subsentence parser can be started on an unparsed symbol to discover additional parsing failures.

As an example, the UML state diagram in Figure 7 contains a syntax error: the two AND-states *NotOn* and *On* are not connected. Thus, if the parsing process starts from the state *High* then only the state *On* and its substates will be parsed. Successively, the bidirectional X-parser can be launched on any unparsed symbol to recognize the remaining symbols.

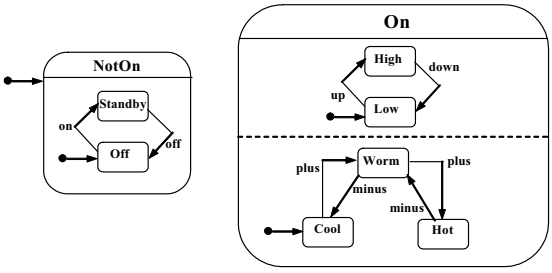

Fig. 7. A global parsing failure in a UML state diagram.

4.2 Exploiting Parsing Information for Symbol Prompting

In the following we describe how to prompt users to insert visual symbols during the composition of a visual sentence. The approach requires the modification of the parsing table construction algorithms in order to extract from the XPG grammar further static information about the relations among the grammar symbols, and the modification of the parsing algorithm in order to associate dynamic parsing context information to each analyzed symbol. By joining the dynamic and static information related to an edited symbol the parser is able to determine all the possible related symbols.

In the next subsection we introduce some preliminary definitions that will be useful for showing the proposed method. Subsection 4.2.2 describes a sketch of the proposed approach to suggest the continuation in the editing of visual sentences.

4.2.1 Preliminary definitions

In the following we introduce the notion of *extended item* that is used by the proposed technique to capture information about relations between visual symbols during the construction of the parsing table. We start by providing some preliminary definitions.

Definition 4.1 Let us consider a production p of the form:

$$A \rightarrow x_1\mathbf{R_1}x_2\mathbf{R_2}\ldots x_{m-1}\mathbf{R_{m-1}}x_m, \Delta, \Gamma \quad \text{for} \quad m \geq 1$$

(i) a couple (hM, kN) is a *link* of p if there exists a relation in p that relates the grammar symbols M and N in p through the syntactic attributes h and k, respectively.

(ii) a couple (hA, kN) is a *tie-point* of p, denoted by ($hA = kN$), if the syntactic attribute h of A is synthesized from the syntactic attribute k of N, i.e., the value of hA depends on kN.

(iii) $JSET(p)$ denotes the set of links and tie-points associated to a production p.

As an example, let us consider the XPG DFD presented in subsection 3.1. The link in production 5 is ($1Node$, $1EDGE$); the links in production 2 are ($1DataFD'$, $1EDGE$) and ($2EDGE$, $1Node$). The tie-points in production 2 are ($1DataFD = 1DataFD'$) and ($1PLACEHOLD = 1Node$).

The notion of $JSET$ for a production can be extended to the XpLR items. This will allow us to keep track of the relations among the grammar symbols during the construction of the set of XpLR item collections. This information will be exploited to prompt the insertion of correct symbols during the editing of a visual sentence.

Definition 4.2 Given an item I, the set $IJSET$ of I, denoted by $IJSET(I)$, is defined as follows:

- if I is the item $S' \rightarrow \cdot S$ then $IJSET(I)$=$\emptyset$;
- if I: $A \rightarrow \cdot\alpha$ then $IJSET(I)$=$JSET(A \rightarrow \alpha) \cup$ IJSET(J) where J is the item in the same collection of items of I, such that I is obtained from the closure on J [1];
- if I: $A \rightarrow \alpha R_1 x \cdot R_2\beta$ then $IJSET(I)$ is the IJSET associated to the item J: $A \rightarrow \alpha \cdot R_1 x R_2\beta$ where the syntactic attributes of the grammar symbol x are marked. A marked syntactic attribute k of a grammar symbol x is denoted by $\underline{kx}$.

As an example, let us consider the XPG DFD. A subset of the sets of XpLR item collection with the associated $IJSET$ is shown in Figure 8. In

such figure superscripts are used to distinguish different occurrences of the same symbol in different items.

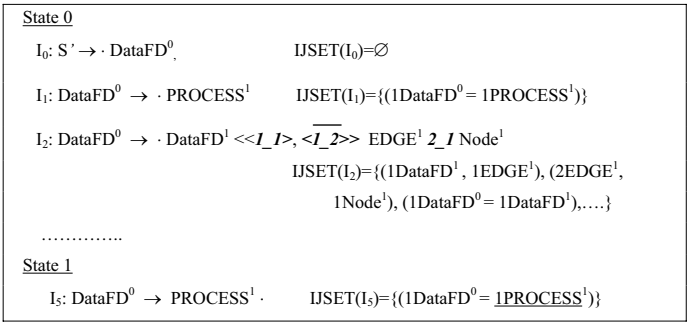

Fig. 8. A subset of the XpLR item collection for the XPG DFD and the associated $IJSETs$.

Adding to each item the corresponding $IJSET$, we have the notion of extended item. In particular, an *extended item* I is a pair $[J, JS]$ where J is an XpLR item and $JS{=}IJSET(J)$.

4.2.2 Symbol prompting

The IJSETs keep track the possible relations between the grammar symbols, and are exploited for prompting the possible symbols that can be related to a particular symbol of a visual sentence. Notice that the IJSETs include information of both driver and tester relations, differently from the next-column entries in the parsing table where only driver relations are stored.

On the other hand, each edited symbol needs the parsing information specifying the context in which it has been recognized. In particular, during the parsing process for each edited symbol we need to keep track of the grammar symbols that synthesize their syntactic attributes. Thus, we associate to each symbol T a set of couples (xGS, s) where GS is the last grammar symbol that has synthesized the syntactic attribute x of T, and s is the state of the forward or backward parser when the symbol GS has been processed. By analyzing the $IJSETs$ of the extended items set I_s of the forward or backward parser we extract the couples $(\underline{xGS}, yNS)$ and consider as symbols to be prompted all the unmarked symbols NS.

As an example, Figure 9 shows a DFD with the reduction of a forward parser. In such a parser the syntactic attribute 1 of $PROCESS$ is synthesized first by $DataFD$ through production 1 then by $DataFD'$ through production 2; whereas the syntactic attribute 1 of $ENTITY$ is synthesized first by the introduced $PLACEHOLD$ symbol through production 2 then by $DataFD'$ through production 4.

The corresponding backward parser simply reduces the $PROCESS$ symbol to another $DataFD$ symbol object.

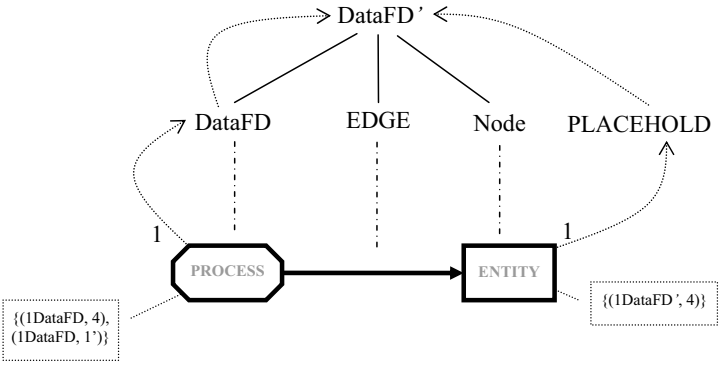

Fig. 9. An example of annotated DFD reduction tree.

Thus, for such couple of forward and backward parsers the $PROCESS$ symbol has associated the two couples $(1DataFD,\ 4)$ and $(1DataFD,\ 1')$, whereas the couple $(1DataFD,\ 4)$ is associated to the $ENTITY$ symbol. The symbols prompted by the forward and backward parsers on the $PROCESS$ symbol are determined by searching in $IJSET(I_4)$ and $IJSET(I_1')$ the unmarked symbols related to $1DataFD$. As shown by the portion of extended item sets I_4 and I_1' in Figure 10 these symbols are $1EDGE$ and $2EDGE$. This means that the start point or the end point of an $EDGE$ symbol can be connected to the borderline of the $PROCESS$ symbol.

State 4	State 1'
I_1 : [DataFD^0 → DataFD^1 · ⟨⟨***I_1***⟩,⟨$\overline{\textbf{\textit{I\_2}}}$⟩⟩ EDGE^1 ***2_1*** Node^1, {(1DataFD^1, **1EDGE^1**), (2EDGE^1, 1Node^1), (1DataFD^0 = 1DataFD^1), (1PLACEHOLD^1 = 1Node^1),….}]	I'$_1$: [S → DataFD^0 ·, {∅}]
I_2 : [DataFD^0 → DataFD^1 · ⟨⟨***I_2***⟩,⟨$\overline{\textbf{\textit{I\_1}}}$⟩⟩ EDGE^1 ***I_1*** Node^1, {(1DataFD^1, **2EDGE^1**), (1EDGE^1, 1Node^1), (1DataFD^0 = 1DataFD^1), (1PLACEHOLD^1 = 1Node^1),….}]	
I_3 : [DataFD^0 → DataFD^1 ⟨***any***⟩ PLACEHOLD^1, {(1DataFD^1, **2EDGE^1**), (1EDGE^1, 1Node^1), (1DataFD^0 = 1DataFD^1), (1PLACEHOLD^1 = 1Node^1),….}]	

Fig. 10. A portion of extended item sets I_4 and I_1'.

5 Conclusions and Future Work

We have presented a strategy for constructing syntax-aware visual language editors. The approach relies on the grammar formalism of XPGs and an incremental LR-based subsentence parsing technique. Once integrated into a visual editor, the parser is able to provide immediate feedback to the users during the composition of visual sentences by highlighting correct and incorrect subsentences, and offering additional support in the construction of the sentences. Indeed, such approach has been exploited for introducing a non-correcting error recovery strategy, and for prompting during the editing the continuation of what the user is drawing.

In order to prove the effectiveness of the presented strategy, also from a usability point of view, we are extending the VLDesk system [4] to support the proposed syntax-aware editing style. Moreover, besides the development of effective graphical interaction of error handling and symbol prompting, an important issue to be addressed in future work is the capability of prompting symbols that corrects the parse failure identified.

References

[1] Aho, A.V., R. Sethi and J.D. Ullman, "Compilers, principles, techniques and tools", Addison-Wesley, 1985.

[2] Bardohl, R., "A Visual Environment for Visual Languages", *Science of Computer Programming* **44** (2), 2002, 181–203.

[3] Chok, S., and K. Marriott, "Automatic Generation of Intelligent Diagram Editors", *ACM Transactions on Computer-Human Interaction* **10** (3), 2003, 244–276.

[4] Costagliola, G., A. De Lucia, R. Francese, M. Risi, G. Scanniello, "A Component-Based Visual Environment Development Process", in *Proceedings of 14th International Conference on Software Engineering and Knowledge Engineering*, Ischia, Italy, 2002, 327–334.

[5] Costagliola, G., A. De Lucia, S. Orefice and G. Tortora, "A Parsing Methodology for the Implementation of Visual Systems", *IEEE Transactions on Software Engineering* **23** (12), 1997, 777–799.

[6] Costagliola, G., and V. Deufemia, "Visual Language Editors based on LR Parsing Techniques", in *Proceedings of SIGPARSE/ACL 8th International Workshop in Parsing Technologies*, Nancy, France, 2003, 79–90.

[7] Costagliola, G., and G. Polese, "Extended Positional Grammars", in *Proceedings of 2000 IEEE Symposium on Visual Languages*, Seattle, WA, USA, 2000, 103–110.

[8] Köth, O., and M. Minas, "Generating Diagram Editors Providing Free-Hand Editing as well as Syntax-Directed Editing", in *Procs. GRATRA'2000 - Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, March 2000, 32-39.

[9] Marriott, K., and B. Meyer, (Eds.) "Visual Language Theory", Springer-Verlag, New York Inc., 1998.

[10] Minas, M., "Concepts and Realization of a Diagram Editor Generator based on Hypergraph Transformation", *Science of Computer Programming* **44** (2), 2002, 157–180.

[11] Tomita, M., "Efficient Parsing for Natural Languages", Kluwer, Boston, 1985.

[12] Tomita, M., "Generalized LR Parsing", Kluwer, Boston, 1991.

[13] Tuovinen, A.-P., "VILPERT - Visual Language Expert", in *Proceedings of the Sixth Fenno-Ugric Symposium on Software Technology FUSST'99*, Tallinn, Estonia, August 19–21, 1999.

[14] Tuovinen, A.-P., "Practical Error Handling in Parsing Visual Languages", *Journal of Visual Languages and Computing* **11** (5), 2000, 505–528.