World Wide Web: Internet and Web Information Systems, 5, 125–157, 2002

A Query Language for XML Based on Graph Grammars

SERGIO FLESCA, FILIPPO FURFARO and SERGIO GRECO {flesca,furfaro,greco}@si.deis.unical.it
DEIS, Università della Calabria, 87030 Rende, Italy

Abstract

In this paper we present a graphical query language for XML. The language, based on a simple form of graph grammars, permits us to extract data and reorganize information in a new structure. As with most of the current query languages for XML, queries consist of two parts: one extracting a subgraph and one constructing the output graph. The semantics of queries is given in terms of graph grammars. The use of graph grammars makes it possible to define, in a simple way, the structural properties of both the subgraph that has to be extracted and the graph that has to be constructed. We provide an example-driven comparison of our language w.r.t. other XML query languages, and show the effectiveness and simplicity of our approach.

Keywords: XML, query language, graph grammar

1. Introduction

The problem of developing query languages for XML, the emerging new standard for the representation of semistructured data on the Web, has been investigated widely. XML documents are composed of a sequence of nested elements: each element is delimited by a pair of tags giving a formal description of their content and a semantics to the enclosed information. Most of the languages proposed so far are declarative languages [18], although there have been proposals for graphical [11] and procedural languages [17].

In this paper we present a declarative, graphical language, called $\mathcal{XGL}$ (XML Graphical Language), for querying XML data. The basic idea underlying our language is that XML and semistructured data can be represented by means of graphs [1,2,9]; thus, the query problem is basically the extraction of subgraphs and the creation of a new graph. It is widely accepted that, for this kind of query, graphical notations are more natural [13,15].

As with most of the proposed languages for XML, $\mathcal{XGL}$ queries consist of two parts used, respectively, for extracting information from data and restructuring information into novel XML documents. Our language provides (graphical) constructs to express nesting of elements, variables associated with both tags and values, path expressions, grouping, and permits us to collect and integrate information coming from different documents.

With respect to other graphical languages [11,12], the main difference of $\mathcal{XGL}$ is that graphs are defined by means of (extended) graph grammars and the semantics of the language is based on the theory of graph grammars [22]. A graph grammar is a graph rewriting system consisting of a set of rewriting rules (or *productions*). As well as a production of a standard grammar defines how to substitute a non terminal symbol (or a group of symbols) with a string, a production of a graph grammar defines how to replace a node (or an edge) in a graph with a subgraph. A graph grammar defines a class of graphs which have common structural properties (e.g., the class of complete graphs, the class of trees, etc.).

Thus, an $\mathcal{XGL}$ query consists of a set of (graphical) production rules which describe the structural property of the graphs which we want to extract and construct. More specifically, the structure of an $\mathcal{XGL}$ query on a set of XML documents is a set of extended graph grammars. Each (extended) graph grammar is described by means of a sequence of extended production rules describing how graphs can be expanded (each rule says how and under which conditions a node of a given graph can be replaced by a specified graph).

In our opinion, the use of (extended) graph grammar production rules makes the language flexible and usable since they make it possible to describe the structural properties of the graph to be extracted in a simple, compact and intuitive way. We point out that the features of $\mathcal{XGL}$ include several requirements which have been recognized [8,27,36] to be of primary importance for a query language for XML documents: declarativeness (the specification of the query defines the content of the result rather than a strategy for its computation), possibility of "reducing" documents (i.e. extracting whole subportions of a document), restructuring documents, expressing join conditions and expressing path expressions.

The rest of the paper is organized as follows. Section 2 contains a brief, informal description of the language. Section 3 presents preliminary definitions of graphs and graph grammars. Section 4 introduces an extension of graph grammars to query graph-like data. Section 5 shows how extended graph grammars can be used to query XML data. Section 6 presents the language $\mathcal{XGL}$.

2. $\mathcal{XGL}$ in a nutshell

In this section we informally present the $\mathcal{XGL}$ query language. We use a classical XML document containing bibliography entries conforming to the following DTD [18]:

```
<!ELEMENT book (title, author+, publisher)>
<!ATTLIST book year CDATA>
<!ELEMENT title     PCDATA>
<!ELEMENT author    PCDATA>
<!ELEMENT publisher PCDATA>
```

The main features of the language are described by means of the Example 1, where some queries over the document below, called `bib.xml` and graphically represented in Figure 1, are described.

```
<bib>
    <book year="1997">
        <title> A First Course in
                Database Systems </title>
        <author> Ullman </author>
        <author> Widom  </author>
        <publisher> Prentice-Hall </publisher>
    </book>
    <book year="1988">
        <title> Principles of Database and
                Knowledge-Base Systems </title>
        <author> Ullman </author>
        <publisher> Computer Science Press </publisher>
    </book>
    <book year="1999">
        <title> Data on the Web </title>
        <author> Abiteboul </author>
        <author> Buneman </author>
        <author> Suciu  </author>
        <publisher> Morgan Kaufmann </publisher>
    </book>
</bib>
```

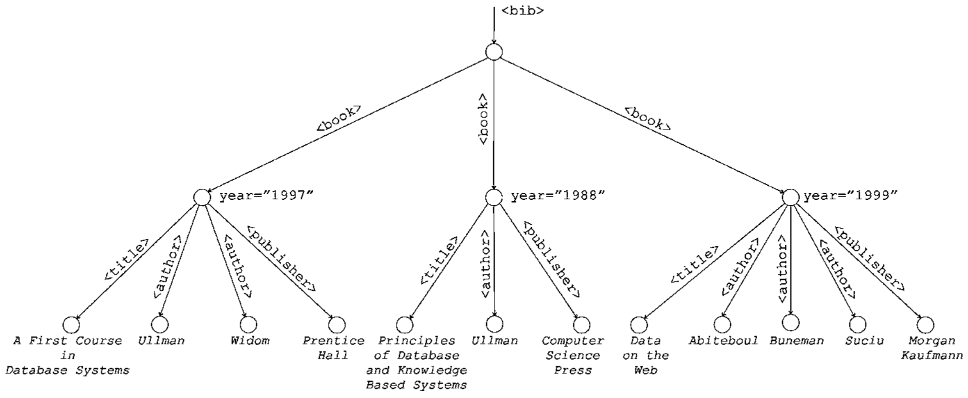

Figure 1. XML graph.

The structure of a $\mathcal{XGL}$ query consists of two parts: one querying and one constructing XML data graphs. The querying part is defined by means of a simplified form of graph grammar, which is made user-friendly by adding some syntactic simplifications to standard graph grammars.

The constructing part is defined by a graph which describes the structure and the content of the document which has to be created.

The querying and constructing parts are correlated by means of variables, which are defined in the querying part (where it is specified what kind of information each variable identifies) and then used in the constructing part (where variables refer to the extracted information).

In the extraction of graphs, graph grammars are coupled with first-order formulas on such variables, in order to express conditions on data and filter them. In particular, every production rule is associated to a (possibly empty) first-order formula which states under which conditions (regarding the data contained in the source graph) the rule can be applied.

Also the constructing rule defining the output graph is associated to a first-order formula, which filters the extracted information and defines how to reassemble it. It is worth nothing that the first-order formulas used in the querying and constructing have different aims: in the querying part the first-order formula is used to select part of the graph whereas in the constructing part it is used to join elements of the graph.

In the representation of graphs of both the querying and constructing part, the language provides shortcuts associated to nodes and arcs. In particular, in order to identify paths in the source graph during the extraction phase, arcs may be labelled with regular expressions defined on a vocabulary of tags. On the other side, nodes may be marked with the symbol "+": such a marked node (called *grouping node*) represents a (possibly empty) set of nodes matching one or more nodes of the input graph.

Like most of the query languages for XML, $\mathcal{XGL}$ queries consist of a "WHERE" clause defining the querying part (extraction of a subgraph) and a "CONSTRUCT" clause defining the constructing part.

The following example presents four queries on the document of Figure 1. Here, each graph grammar consists of only one production rule. We shall use, respectively, the symbol **S** to denote the axiom (start symbol) of the graph grammar used to extract sub-graphs, and the symbol **T** to denote the constructing rule which defines how to build the output document. Thus, the form of all the queries presented in the following examples will be:

```
WHERE S IN "bib.xml"
CONSTRUCT T
```

Example 1.

1. *Construct a document containing the titles of all books printed by Prentice-Hall from 1992 on.* We first extract for each book the pairs $\$t/\$n$, where $\$t$ denotes the title and $\$n$ the publisher of the book satisfying the condition that the year of the book is after 1991 and the publisher is "Prentice-Hall" ($b.year > 1991 ∧ $n.value = "Prentice-Hall"). The symbol "+" inside the node labelled with $\$b$ means that we are interested in all books. In the matching with the input graph, the node is expanded into a list of nodes to match a maximal number of nodes. The output graph is constructed by means of the constructing rule denoted by the symbol **T** on the right side of Figure 2. Here we construct an XML document having a structure similar to that of the input document, but where each book contains only the attribute *year* and the element *title*. The symbol "+" inside the node ending the arc with label <book> means that we are interested in all books and, therefore, the output graph may contain more arcs with tag <book>. The variables $\$b$ and $\$t$ are used to pass data from the input graph to the output graph. The condition arc($b, $t, <title>) states that we consider pairs ($\$b$, $\$t$) which in the extracted graph are connected by an arc with label <title>.
2. *Construct a document containing for each book and for each author of the book, the pairs* (*title*, *author*). The query reported in Figure 3 extracts, for each book, the title and the set of all authors. The symbol "+" inside the node ending the arc with tag

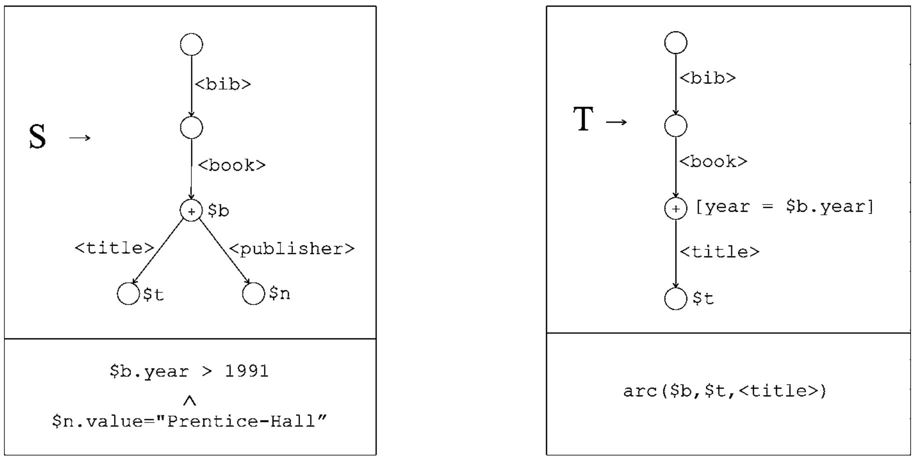

Figure 2. $\mathcal{XGL}$ query 1.

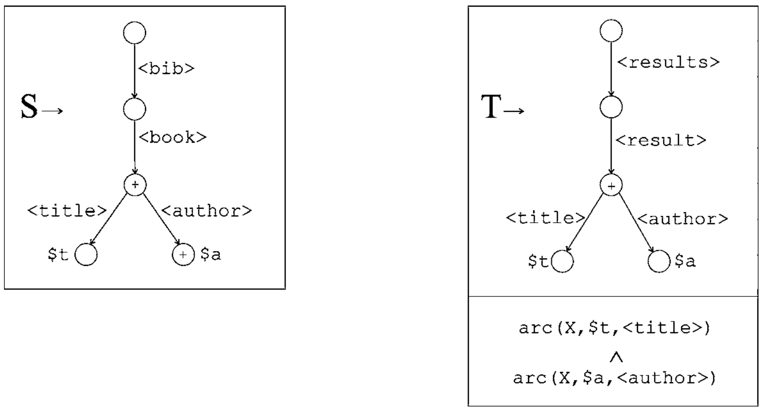

Figure 3. $\mathcal{XGL}$ query 2.

`<book>` means that we are interested in all books, and the same symbol marking the nodes at the end of the arc with tag `<author>` means that we want to collect, for each book, all of its authors. Thus, a set of results is constructed and each result contains a pair $\$t/\$a$, where $\$t$ denotes a title of book and $\$a$ an author. The condition `arc(X, $t, <title>)` $\wedge$ `arc(X, $a, <author>)` states that we are only interested in pairs ($\$t$, $\$a$) identifying, respectively, the title and the author of the same book (X).

3. *Construct a document containing, for each book, the title and the set of all authors.* This query can be expressed by a simple variation of the rule **T** in Figure 3. In particular, by putting the symbol "+" inside the node ending the arc marked with `<author>`, we collect for each book all of its authors.
4. *Construct a document containing, for each author, the titles of the books written by him.* Also this query can be expressed by a simple variation of the rule **T** in Figure 3.

In particular, by putting the symbol "+" inside the node ending the arc marked with `<title>`, we collect for each author all the titles of the books he has written.

3. Preliminaries

3.1. *Data graphs*

Let Γ be an alphabet of node labels and Σ an alphabet of edge labels. A graph over Γ and Σ is a tuple $D = (N, E, \lambda)$ where N is a set of nodes, $E \subseteq \{(u, \sigma, v) | u, v \in N, \sigma \in \Sigma\}$ is a set of labelled edges and $\lambda : N \to \Gamma$ is a node labelling function. We identify a subset $\Delta \subseteq \Gamma$ as the set of *terminal* node labels. A node x of a graph D is said to be terminal if $\lambda(x) \in \Delta$ and we say that D is terminal if all its nodes are terminal. An arc from u to v with label σ is denoted by $u \xrightarrow{\sigma} v$. The components of an edge e will be denoted by $e[1]$, $e[2]$ and $e[3]$, respectively.

A path over D is a sequence $p = (v_1, e_1, v_2, e_2, \ldots, v_n)$ where $v_i \in N$, $e_j \in E$, $e_i[1] = v_i$ and $e_i[3] = v_{i+1}$. The label path of p, denoted *label*(p), is a subset of Σ^* defined as $e_1[2] \cdots e_{n-1}[2]$. Given a regular expression r over Σ and a string $w \in \Sigma^*$, we say that w spells a path p in D if $w =$ *label*(p) and we say that p satisfies r if *label*$(p) \in \mathcal{L}(r)$, i.e. the string spelled by p belongs to the language defined by r.

Given two data graphs A and B, we say that A is a subgraph of B iff (i) $N_A \subseteq N_B$, (ii) $\forall x \in N_A \ \lambda_A(x) = \lambda_B(x)$, and (iii) $\forall x, y \in N_A$, $(x, \sigma, y) \in E_A$ only if $(x, \sigma, y) \in E_B$.

3.2. *NR graph grammars*

Graph grammars generalize standard grammars and *context-free* graph grammars are the natural generalization of context-free grammars: standard grammars generate strings whereas graph grammars generate graphs [19]. Two main types of context-free graph grammars have turned out to be the most natural, robust, and easy to handle: the *Hyper-edge Replacement (HR) grammars* and *Node Replacement (NR) grammars*. In this paper we consider NR context-free graph grammars.

Node Replacement grammars generate labelled, directed graphs. A production of a graph grammar is of the form $X \to (D, C)$ where X is a nonterminal node label, D is a graph and C is the set of connection instructions. A rewriting step of a graph H according to such a production consists of removing a node u labelled X from H, adding D to H and adding edges between D and H as specified by the connection instructions in C. The pair (D, C) can be viewed as a new type of object, and the rewriting step can be viewed as the substitution of the object (D, C) for the node u in the graph H. Intuitively, these objects are quite natural: they are graphs ready to be embedded in an environment. Their formal definition is as follows.

Let Γ be an alphabet of node labels and Σ an alphabet of edge labels. A *graph with embedding* is a pair $K = (H, C)$ where H is a graph over Γ and Σ and $C \subseteq \Gamma \times \Sigma \times \Sigma \times N \times \{in, out\}$ is the connection relation of K. Each element $(\gamma, \sigma_1, \sigma_2, v, d) \in C$ is a connection instruction of K and is generally written as $(\gamma, \sigma_1/\sigma_2, v, d)$. The components of a graph with embedding K will be denoted as N_K, E_K, λ_K and C_K.

Intuitively, for a graph with embedding K, the meaning of a connection instruction $(\gamma, \sigma_1/\sigma_2, v, out)$ is as follows: if there was a σ_1-labelled edge from a node u which has been substituted by K to a γ-labelled node w, then the embedding mechanism defines a σ_2-labelled edge from v to w. Similarly, the meaning of a connection instruction $(\gamma, \sigma_1/\sigma_2, v, in)$ is as follows: if there was a σ_1-labelled edge from a γ-labelled node w to a node u which has been substituted by K, then the embedding mechanism defines a σ_2-labelled edge from w to v. The feature which replaces edge labels is called *dynamic edge labelling*.

Let H be a graph over Γ and Σ, K be a graph with embedding over the same alphabets, and let $v \in N_H$. The substitution of K for v in H is denoted by $H[v/K]$.

In the following, connection rules of the form $(\gamma, \sigma/\sigma, v, a)$ (i.e. rules which do not relabel edges) are simply written as (γ, σ, v, a).

Definition 1. A *node replacement* (NR) grammar is a tuple $G = (\Gamma, \Delta, \Sigma, P, S)$ where Γ is the alphabet of node labels, $\Delta \subseteq \Gamma$ is the alphabet of terminal node labels, Σ is the alphabet of edge labels, P is the finite set of productions, and $S \in \Gamma - \Delta$ is the initial nonterminal symbol (axiom). A production is of the form $X \to (D, C)$ where $X \in \Gamma - \Delta$ and (D, C) is a graph with embedding over the alphabets Γ and Σ.

Example 2. The grammar G defined by the productions shown in Figure 4 describes a language containing chains. Connection rules can be incorporated into the left and right parts of production rules.

Figure 5 illustrates a chain derivation by means of G productions.

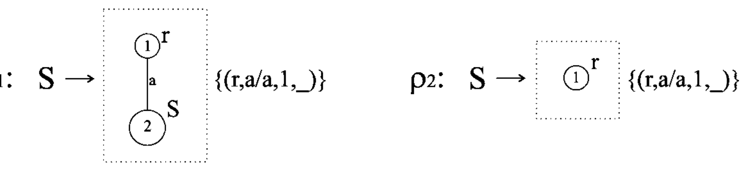

Figure 4. A graph grammar producing chains.

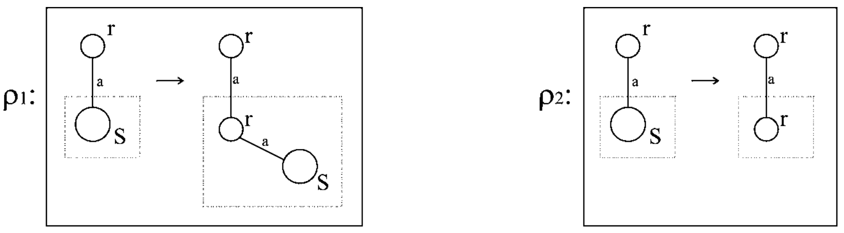

Figure 5. A graph grammar equivalent to that of Figure 4.

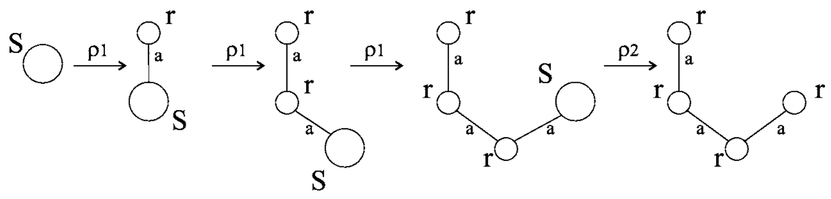

Figure 6. Derivation of a chain.

The graph appearing in the right side of a production can be empty and a production of the form $X \to (\emptyset, \emptyset)$ will be simply denoted as $X \to \varepsilon$.

Let $G = (\Gamma, \Delta, \Sigma, P, S)$ be an NR grammar. Let H and H' be two graphs, let $v \in N_H$ and let $p : X \to (D, C)$ be a production of G. Then, we say that H' is directly derived from H (and write $H \Rightarrow_{v,p} H'$, or just $H \Rightarrow H'$), if $\lambda_H(v) = X$ and $H' = H[v/(D, C)]$. Moreover, we say that H' is derived from H if there is a finite sequence $H \Rightarrow H_1 \Rightarrow \cdots \Rightarrow H'$.

A graph grammar G defines a class of graphs which have common structural properties. The set of graphs generated by G is called *graph language* and denoted as $\mathcal{L}(G)$.

4. Querying data graphs

We start by defining a simple graph model on an alphabet with three different types of symbol labelling nodes: constants, variables and nonterminal symbols. A variable can take any value and, therefore, it can be associated to any constant. In the following, constants are represented by strings starting with digits or lowercase letters (e.g., $b1$), variable names are denoted by strings preceded by a dollar (e.g., $\$b1$) and nonterminal symbols are denoted by strings starting with uppercase letters (e.g., X). Given an alphabet of node labels Γ we will denote with Γ_c, Γ_v and Γ_{nt} the subsets of Γ which contain, respectively, constant symbols, variable symbols and nonterminal symbols. A graph over the two alphabets $\Gamma = \Gamma_c \cup \Gamma_v \cup \Gamma_{nt}$ and Σ will be called *query graph*. A query graph whose nodes are labelled only with constant symbols (i.e. $\Gamma = \Gamma_c$) will be called *data graph*. A query graph which does not contain any nonterminal nodes (i.e. $\Gamma_{nt} = \emptyset$) is called *terminal query graph*.

Thus, data graphs only contain constants and are used to represent the input database; terminal query graphs are used to denote graphs which can be “mapped” on data graphs (by associating variables appearing in the terminal query graph to the constants labelling nodes of the data graph, or by associating nodes of the terminal query graph labelled with a constant to nodes of the data graph labelled with the same constant). General query graphs are used to represent the intermediate steps of the derivation of the query graphs obtained applying graph grammars.

Given a query graph α, we shall denote with *Terminal*(α) the subgraph derived from α by deleting nodes marked with nonterminal symbols and arcs connected to deleted nodes.

Example 3. The graph grammar G consisting of the productions of Figure 7 defines a language consisting of trees. Figure 8 illustrates a tree derivation by means of G produc-

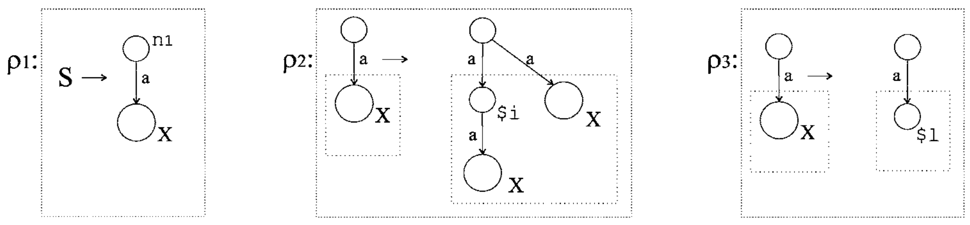

Figure 7. Graph grammar defining trees.

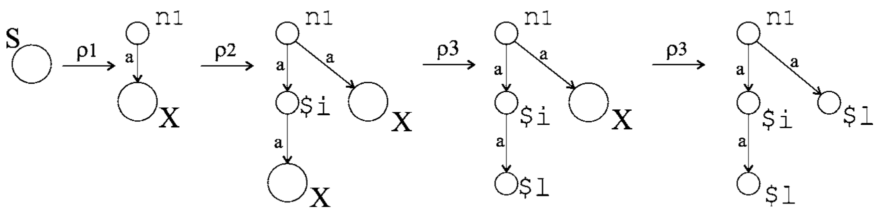

Figure 8. Graph derivation.

tions. Note that the root nodes of the trees in the language defined by this grammar have a specific data label (n_1), the internal nodes have label $\$i$ and the leaf nodes have label $\$l$.

Since in this context we are not interested in generating new graphs, but only in identifying subgraphs of a given data graph, we shall not consider the whole language generated by a grammar, but only a subset containing graphs which identify some portion of the input data graph. To this purpose we define a mapping from terminal query graphs (obtained at the end of a graph grammar derivation) to subgraphs of a given data graph.

Definition 2. Let $\alpha = (N, E, \lambda)$ be a terminal query graph over Γ and Σ, and $D = (N_D, E_D, \lambda_D)$ a data graph over Γ_c and Σ. A *mapping* φ from α to D is a total function mapping, respectively, nodes in N to nodes in N_D and edges in E to edges in E_D such that (i) for each node $n \in N$ either $\lambda(n) = \lambda(\varphi(n))$ or $\lambda(n)$ is a variable label, (ii) for each arc $(u, \sigma, v) \in E$ there is an arc $(\varphi(u), \sigma, \varphi(v)) \in E_D$, and (iii) there are no two nodes u and v such that $\lambda(u) = \lambda(v)$ and $\varphi(u) = \varphi(v)$ (i.e. two nodes with the same label cannot by associated to the same node in D).

Definition 3. Let D be a data graph. A *mapping pair* on D is a pair (α, φ) where α is a query graph and φ is a data mapping from *Terminal*(α) to D.

Observe that *Terminal*(α) is a terminal query graph (i.e. a graph whose node labels can be either constants or variables). Moreover, a mapping pair (α, φ) is said to be *terminal* if α is a terminal query graph. Like an embedded graph, a mapping pair can be seen as a new type of object consisting of a query graph (derived from a graph grammar) mapped over a given data graph. The derivation of query graphs from parsing grammars can be extended to mapping pairs. Let D be a data graph, G a graph grammar and (α, φ) a mapping pair over D. We say that a mapping pair (β, ψ) is directly derived from (α, φ) through a production ρ of PG (and write $(\alpha, \varphi) \Rightarrow^{\rho} (\beta, \psi)$) if and only if $\alpha \Rightarrow^{\rho} \beta$ and ψ extends φ (i.e. $\varphi \subseteq \psi$). Moreover, we say that a mapping pair (α_n, φ_n) is derived from a mapping pair (α_0, φ_0) over a data graph D if $(\alpha_0, \varphi_0) \Rightarrow^{\rho_1} (\alpha_1, \varphi_1) \Rightarrow^{\rho_2} \cdots \Rightarrow^{\rho_n} (\alpha_n, \varphi_n)$. Given a graph grammar G and a data graph D, $\Phi(G, D)$ defines the set of terminal mapping pairs derived from $(S, \emptyset)$ where $\emptyset$ denotes an empty mapping.

A terminal mapping pair applied to a data graph D allows us to identify a subgraph of D having the property defined by the grammar. Each node of the extracted subgraph can be associated to more than one node of the query graph, if these nodes have different labels (roles). Different labels are used to distinguish different classes of nodes (e.g., in a tree internal nodes and leaf nodes may have different labels).

Example 4. Consider the parsing grammar of Example 2, the derivation shown in Example 2 and the data graph shown in Figure 9.

The query graphs produced respectively at the third and last steps of the derivation can be mapped on D as shown in Figure 10.

Note that the production defining the axiom (start symbol of the graph grammar) contains an arc whose source node is marked with the constant label $n1$. This means that all derived query graphs are trees whose root node is marked with $n1$. Therefore, every tree generated by such grammar can be mapped only to a tree whose root node has label $n1$. In the above mapping $\lambda(1) = \lambda(\varphi(1)) = n1$ whereas all other nodes in the query graph have associated a variable. Although not represented in the figure, the arcs in the query graph

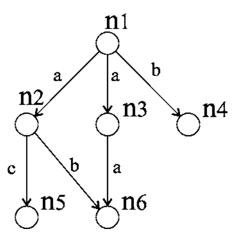

Figure 9. A data graph.

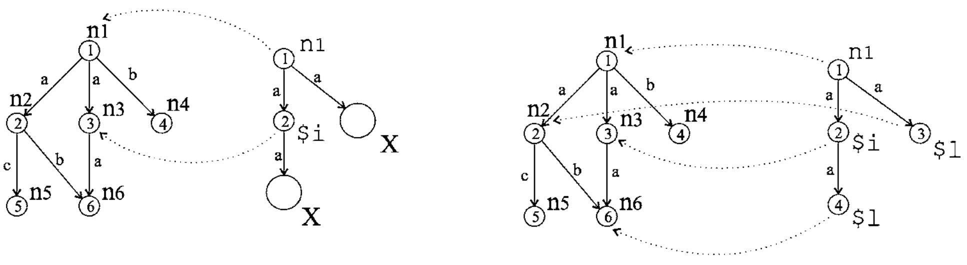

Figure 10. Mapping of query graphs to a data graph.

are mapped to arcs in the data graph; for instance, the arc $e = (1, a, 2)$ and the arc $\varphi(e)$ have the same label a (i.e. $\varphi(e) = (\varphi(1), a, \varphi(2))$).

Parsing grammars. We now introduce a new type of graph grammars, called *parsing grammars*, which are specialized in extracting information from data graphs. Parsing grammars have the following characteristics:

- the set of production rules is linearly ordered (in order to drive the derivation process and reduce the nondeterminism);
- a rule can only be applied if a certain condition on the extracted data is satisfied.

Definition 4. A *Parsing (Graph) Grammar* is a tuple $PG = (\Gamma, \Sigma, P, S)$, where Σ is the alphabet of edge labels, Γ is the alphabet of node labels ($\Gamma \cap \Sigma = \emptyset$) and $S \in \Gamma_{\text{nt}}$ is the axiom. P is a linearly ordered set of productions of the form $X \to (\alpha, C, \Theta)$, where

(1) $X \in \Gamma_{\text{nt}}$ is a nonterminal symbol,
(2) α is a query graph over Γ and Σ,
(3) C is a set of connection rules, i.e. a set of tuples (γ, σ, v, d) where $d \in \{in, out\}$, $\gamma \in \Gamma$, $\sigma \in \Sigma$ and v is a node,
(4) Θ is a first-order formula on Σ, Γ_{c}, Γ_{v} without quantifiers,
(5) for each symbol $X \in \Gamma_{\text{nt}}$ there is a production $X \to \varepsilon$ in P,
(6) for each pair of productions $\rho_i : X \to (\alpha, C, \Theta)$ with α not empty and $\rho_j : X \to \varepsilon$ is $\rho_i < \rho_j$,
(7) for each production $\rho : X \to (\alpha, C, \Theta)$ with α not empty, D contains at least one terminal node.

Thus, a parsing grammar is a restricted form of graph grammar. The restrictions have been introduced to reduce the nondeterminism (items (5)–(7) in Definition 4). The formula Θ states under which condition the rule can be applied.

Parsing grammars generate terminal query graphs without allowing edge relabelling. The formal semantics of production rules can be done by extending the definition of the derivation of mapping pairs.

Let (β, φ) be a mapping pair, $\rho : X \to (\alpha, C, \Theta)$ a parsing grammar production rule and $\rho' : X \to (\alpha, C)$ the corresponding "standard" rule. We say that a mapping pair (γ, ψ) directly derives from (β, φ) through ρ (written $(\beta, \varphi) \Rightarrow^{\rho} (\gamma, \psi)$) if $(\beta, \varphi) \Rightarrow^{\rho'} (\gamma, \psi)$ and Θ is true w.r.t. the pair (α, ψ).[1] The formula Θ is true w.r.t. (α, ψ) if by replacing the variables in Θ with the constants associated with ψ the resulting formula is satisfied.

The order of the productions of a parsing grammar *PG* defines an order on the mapping pairs derived from *PG*. Given a data graph D, a parsing grammar *PG*, and two productions ρ and ν of *PG* such that $\rho < \nu$, we say that a derivation d_1 of a pair (α_1, φ_1) from a pair (α, φ) precedes a derivation d_2 of a pair (α_2, φ_2) from (α, φ) (written $d_1 \prec d_2$), if (1) $d_1 = (\alpha, \varphi) \Rightarrow^{\rho} (\alpha_i, \varphi_i) \Rightarrow^{*} (\alpha_1, \varphi_1)$, $d_2 = (\alpha, \varphi) \Rightarrow^{\nu} (\alpha_j, \varphi_j) \Rightarrow^{*} (\alpha_2, \varphi_2)$, or (2) there are three derivations d, d_3 and d_4 such that $d_1 = dd_3$ and $d_2 = dd_4$ and $d_3 \prec d_4$.

We can now use the relation $\prec$ to define a partial order on the set of derived mapping pairs $\Phi(PG, D)$. Given two mapping pairs $M_1, M_2 \in \Phi(PG, D)$, we say that $M_1 <_{PG} M_2$

if for each derivation $d_2 = (S, \emptyset) \Rightarrow^* M_2$, there exists a derivation $d_1 = (S, \emptyset) \Rightarrow^* M_1$ such that $d_1 \prec d_2$. The order introduced on the productions of *PG* makes $\Phi(PG, D)$ partially ordered. A mapping pair $M \in \Phi(PG, D)$ is said to be *minimal* if there is no mapping pair $M' \in \Phi(PG, D)$ such that $M' <_{PG} M$.

Theorem 1 [23]. Let *PG* be a parsing grammar and *D* a data graph. The nondeterministic selection of a minimal mapping pair in $\Phi(PG, D)$ can be computed in polynomial time.

Clearly, any mapping pair in $\Phi(PG, D)$ (not necessarily a minimal one) selected nondeterministically can also be computed in polynomial time.

The above theorem states that the extraction of a subgraph in the class of graphs defined by the parsing grammar can be done efficiently.

5. Graph grammars for XML data

In this section we present a data model for representing XML documents by means of graphs, and then specialize parsing grammars to extract data from XML graphs.

An XML document can be represented as an ordered, labelled and oriented graph where:

- the containment relation between two elements is represented by an arc labelled with the tag of the subelement;
- references are represented by arcs connecting the referencing element to the referenced one, and such arcs are labelled with the name of the reference attribute;
- each node contains the set of the attributes of the corresponding element;
- if an element contains text and does not contain any subelements, the text is assimilated to the value of an attribute *value*;
- if an element contains both subelements and text strings, each string is assimilated to the attribute *string* = *"string-value"* of a subelement `<text>`.

The representation that we adopt in this paper can be easily understood by examining the document and the corresponding graph of Example 5.

Example 5. Figure 11 shows an XML document containing IDREFs. The associated graph contains arcs of different types which could both be navigated.

Observe that the dotted arcs denote attributes whereas solid arcs denote elements.

In the following definition we formally identify the structure of an *XML Graph*.

Definition 5. Let *A* be a set of attribute names, *T* a set of tag names, and *V* a set of attribute values. An unordered XML graph is a labelled oriented graph $G = \langle N, E_r \cup E_t, f, r\rangle$ where:

- N is the set of nodes,
- $E_{\mathrm{r}} \subseteq \{(u, \sigma, v) | u, v \in N \text{ and } \sigma \in A\}$ is a set of reference arcs,

```
<db>
  <writer ID="1"> Name1 </writer>
  <writer ID="2"> Name2 </writer>
  <writer ID="3"> Name3 </writer>
  <book author="1"> Title1 </book>
  <book author="3"> Title2 </book>
</db>
```

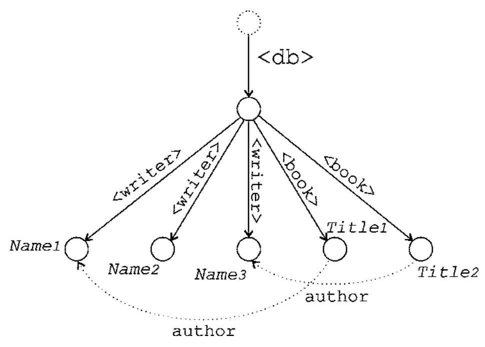

Figure 11. Example of XML graph.

- $E_t \subseteq \{(u, \sigma, v) | u, v \in N \text{ and } \sigma \in T\}$ is a set of tag arcs,
- $f : N \to 2^{A \times V}$ is the function associating a set of attribute/value pairs to each node,
- $\langle N, E_t \rangle$ is a tree with root r.

An ordered XML graph is a quintuple $G = \langle N, E_r \cup E_t, f, g, r \rangle$ where N, E_r, E_t, f and r are defined as above and $g : E_t \to Z^+$ is a function associating an unique ordinal number to each arc in E_t.

In the previous sections, we showed that parsing grammars can be used to extract information from a source data graph. Now we "specialize" parsing grammars to extract information from XML graphs.

The only difference with respect to the querying of graphs introduced in the previous section is that each node in an XML graph has a set of attributes and a value. Attributes and values can be identified by using a "dot" notation as shown in the examples of Section 2. Thus, the attribute *year* of the element *book* identified by the variable $\$b$, is denoted by $\$b.year$. The text contained in the element $\$b$ is denoted by $\$b.value$.

The following example shows how XML subgraphs can be extracted from XML documents.

Example 6. A parsing grammar extracting from the document described in Section 1 the titles of the books published after 1991 by Prentice-Hall (equivalent to the $\mathcal{XGL}$ parsing grammar of Example 1) is reported in Figure 12.

The characterization of graphs given in Section 4 can be easily extended to XML-like graphs. Thus, XML data graphs are XML graphs whose labels are constants, XML query graphs may have both constants and variables as terminal labels, and may contain non terminal labels. For instance, the graphs used by production rules (see Figure 12) are XML query graphs.

The derivation process of an XML parsing grammar applied to a source XML data graph *XD* leads to a terminal mapping pair (α, φ) where φ associates each variable of α to a set of nodes of *XD*.

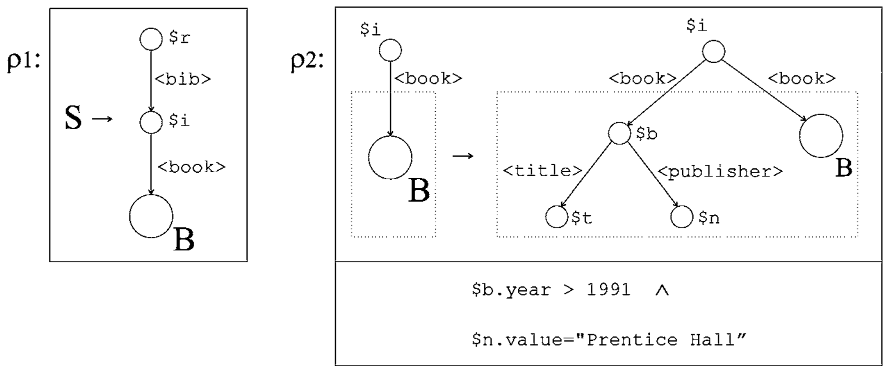

Figure 12. XML parsing grammar.

6. $\mathcal{XGL}$: A graphical language for XML

XML parsing grammars can be used for extracting data from XML documents. Here, we further extend graph grammars for extracting data making them more user-friendly, and introduce simple construction rules to restructure information into new documents. Thus, we present the language $\mathcal{XGL}$ which is a graphical language derived from the grammars introduced in the previous section by adding new features to simplify the process of extracting subgraphs and to construct the output graph.

An $\mathcal{XGL}$ query is of the form[2]

```
WHERE        $S_1$ IN $F_1, \ldots, S_n$ IN $F_n$
CONSTRUCT $T$ [ ( $\$u_1 \rightarrow \$v_1, \ldots, \$u_k \rightarrow \$v_k$ ) ] [ AS   $F_0$ ]
```

where $F_0, \ldots, F_n$ are file names, $S_1, \ldots, S_n$ are axiom symbols corresponding to $\mathcal{XGL}$ parsing grammars, T is an $\mathcal{XGL}$ constructing rule and $\$u_j \rightarrow \$v_j$ means that the variable $\$u_j$ appearing in the WHERE clause is renamed as $\$v_j$ in the CONSTRUCT clause.

Thus, the WHERE clause is used to extract and mark information from the specified XML documents, whereas the CONSTRUCT clause specifies how to reorganize the extracted information in the XML document which results.

In the following, given an $\mathcal{XGL}$ query Q and a set of XML documents $D_1, \ldots, D_k$, $Q(D_1, \ldots, D_k)$ denotes the set of all documents which can be constructed by applying Q to $D_1, \ldots, D_k$.

6.1. $\mathcal{XGL}$ parsing grammars

Definition 6. An $\mathcal{XGL}$ parsing grammar is an XML parsing grammar where:

(1) nodes may be marked with the symbol "+"; these nodes, called *grouping nodes*, denote sets of nodes labelled with the same symbol;
(2) arcs may be labelled with general regular expressions denoting not empty strings.

Moreover, if the regular expression associated to a given arc contains the union symbol " |" or the closure symbols "$\star$" and "$+$", the ending node must be a grouping node labelled with a terminal symbol.

Observe that the restriction on arcs with regular expressions has been introduced to avoid ambiguity in the result. General regular expressions denoting not empty strings may be rewritten into regular expressions without ε and $\star$ symbols. Therefore, in the following we assume that our regular expression denoting not empty paths does not contain both symbols ε and $\star$.

The formal semantics of $\mathcal{XGL}$ parsing grammars can be done in terms of XML parsing grammars by defining how production rules containing shortcuts (grouping nodes and arcs labelled with regular expressions) are rewritten into XML production rules. We first show how production rules with regular expressions are rewritten, and next consider the rewriting of production rules with grouping nodes.

Each rule ρ containing shortcuts is rewritten into one or two standard rules denoted, respectively, by r and r_1, r_2. For the sake of simplicity, in our rewriting rules we do not consider the contexts associated with the rules, and assume that each node labelled with X in $r_1, \ldots, r_k$ has the same context of the node marked with X in ρ.

6.1.1. Rewriting of productions with regular expressions. A production rule with a regular expression is rewritten into a set of equivalent rules as follows:

- *Concatenation.* The rewriting of a production rule containing an arc labelled with the concatenation $p.q$ is reported in Figure 13 where the rule ρ is replaced by the rule r. Observe that there are two different rewritings, respectively, for arcs ending with simple nodes and grouping nodes.
- *Union.* The rewriting of a production rule containing an arc labelled with the union $p \mid q$ is shown in Figure 14 where the rule ρ is replaced by the rule r.

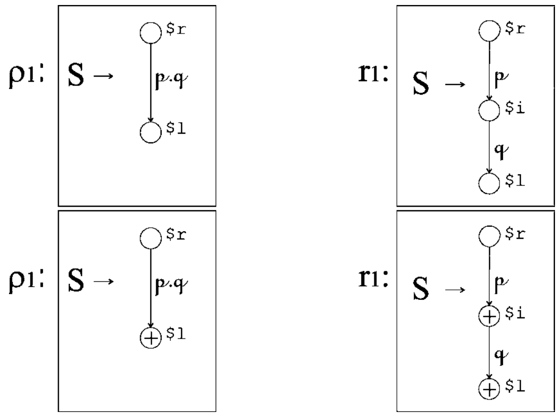

Figure 13. Rewriting arcs with concatenation.

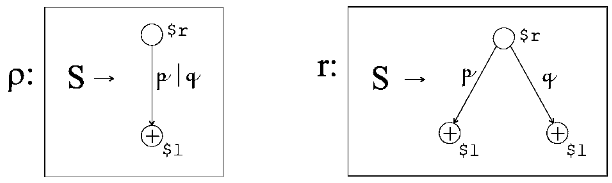

Figure 14. Rewriting arcs with union.

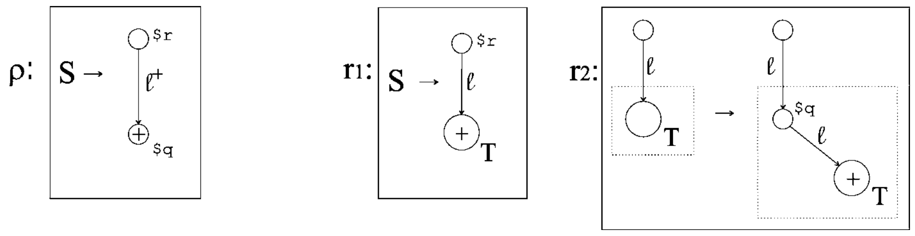

Figure 15. Rewriting arcs with closure.

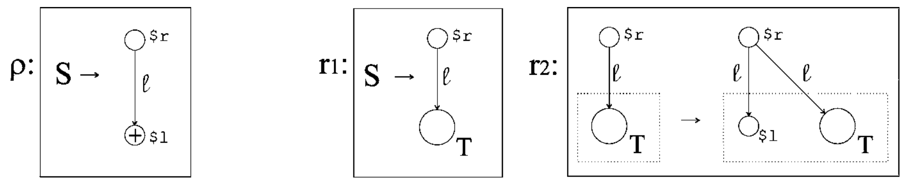

Figure 16. Rewriting grouping nodes.

- *Closure.* The rewriting of a production rule containing an arc labelled with the positive closure l^+ is shown in Figure 15 where the rule ρ is replaced by the two rules r_1 and r_2.

6.1.2. Rewriting of productions with grouping nodes. The rewriting of a production rule containing grouping nodes is shown in Figure 16 where the rule ρ is replaced by the two rules r_1 and r_2 and T is a new nonterminal symbol.

Example 7. The rewriting of the parsing grammar in the first query of Example 1 (Figure 2) is reported in Example 6 (Figure 12). The rewriting of the parsing grammar in the second query of Example 1 (Figure 3) is given in Figure 17.

6.2. $\mathcal{XGL}$ constructing rules

In the previous section we have shown that $\mathcal{XGL}$ parsing grammars can be used to extract information from an XML data graph, allowing us to specify the structure of the subgraph containing such information.

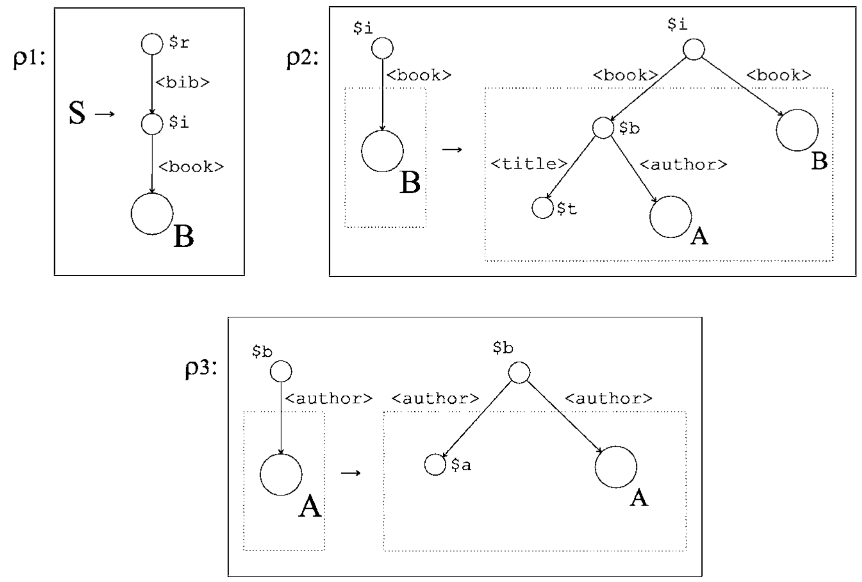

Figure 17. Rewriting of the $\mathcal{XGL}$ parsing grammar of Example 1 (query 2).

In this section we now show that in a similar way, using the graph containing the extracted information, it is possible to build a new document. The process of creating a new XML graph is carried out by defining a tree structure containing information from the extracted graph which satisfies a given condition. The new graph is defined by means of an $\mathcal{XGL}$ constructing rule whereas the condition is defined by means of a first-order formula.

Syntax. The first-order formula is based on the ternary predicates *arc* and *path* storing, respectively, information about arcs and paths in the extracted graph. The arguments of the predicates *arc* and *path* may be constants, variables representing node labels (denoted by lowercase strings preceded by \$) and *general variables* representing node identifiers, and edge labels (denoted by strings starting with an uppercase letter).

For instance, the fact `arc(X, $t, < title >)` states that there is an arc labelled `< title>` from a generic node to a node labelled $\$t$. Analogously, the fact `path(X, $t, < title> *)` states that there is a path whose first arc is labelled `<title>` from a generic node to a node labelled $\$t$.

Definition 7. Let D be an XML data graph, (β, φ) a terminal mapping pair on D. An *$\mathcal{XGL}$ constructing rule* has the form $T \twoheadrightarrow (\alpha, \Theta)$, where α is an XML query graph, and Θ is a *FO* formula on $\{arc, path\}$ evaluated on D.

It is worth noting that instead of *FO* it is possible to use alternative languages such as *SQL*, logic languages or graphical notations.[3]

Semantics. We now present the semantics of constructing rules. At the end of the parsing process we have produced a mapping pair (β, φ) over an XML data graph D, where each variable symbol $\$v$ in β is associated to a (possibly empty) set of nodes in D. The association between variables and nodes is represented by means of a ternary relation *Node* where a tuple $(id, val, \$v)$ states that a node in β labelled with $\$v$ is associated (by φ) to the node with identifier *id* and value *val* in D.[4] Moreover, we assume that the graph D is stored by means of the ternary predicate *Arc* which is different from the predicate *arc* used in the *FO* formula: *Arc* corresponds to the set E_D (i.e. $Arc(id_1, id_2, l)$ is true iff $(id_1, l, id_2) \in E_D$). Analogously, the predicate *Path* denotes the transitive closure of *Arc*, and corresponds to the predicate *path* which defines the transitive closure of *arc*.

The query graph α, in a constructing rule (α, Θ), defines the structural properties of the output XML document, while the first-order formula Θ expresses a condition defining which nodes in the data graph extracted in the `WHERE` clause will be used in the construction of the output graph.

Observe that in the *FO* formula Θ, the predicates *arc* and *path* take as arguments (i) variables representing node identifiers and edge labels, denoted by capital letters (e.g., X), (ii) variables representing node labels, denoted using the symbol \$ (e.g., $\$x$), and (iii) constants. To explain the semantics of the construction rule we first rewrite the condition Θ, translating the predicates *arc* and *path* in an equivalent formula containing only the predicates *Arc* and *Path* in order to obtain a formula Θ' which is directly verifiable on D. Before introducing the formal rewriting we present an example.

Example 8. Consider the construction rule of the second query in Example 1. The condition

$$\texttt{arc(X, \$t, <title>)} \wedge \texttt{arc(X, \$a, <author>)}$$

is rewritten as

$$\texttt{Node(Id}_\texttt{a}\texttt{, V}_\texttt{a}\texttt{, \$a)} \wedge \texttt{Node(Id}_\texttt{t}\texttt{, V}_\texttt{t}\texttt{, \$t)} \wedge \texttt{Arc(X, Id}_\texttt{t}\texttt{, <title>)}$$
$$\wedge\, \texttt{Arc(X, Id}_\texttt{a}\texttt{, <author>)}$$

The variables `$a` and `$t` in the graph are replaced, respectively, by $\texttt{V}_\texttt{a}$ and $\texttt{V}_\texttt{t}$.

The formal rewriting of construction rule $T \rightarrow (\alpha, \Theta)$ into a rule $T \rightarrow (\alpha', \Theta')$ is as follows:

(1) α' is derived from α by replacing every occurrence of $\$x$ with V_x.
(2) Define Θ'' as the first-order formula derived from Θ by replacing (i) all occurrences of the predicate *arc* with *Arc*, (ii) all occurrences of the predicate *path* with *Path* and (iii) every variable $\$x$ with Id_x.
(3) $\Theta' = \bigwedge_{\substack{\$x \ in\ \Theta \\ \$x\ in\ \alpha}} Node(Id_x, V_x, \$x) \wedge \Theta''$.

Using the rewritten constructing rule we obtain the output graph by expanding the grouping nodes and replacing variables with constants satisfying the condition Θ'. The expansion of a grouping node appearing in an $\mathcal{XGL}$ constructing rule can be explained as similar to the expansion of a grouping node in the application of a parsing grammar production. When, in a parsing grammar rule, a node n is marked with the symbol "+" all the (maximal) subtrees rooted in n and matching the specified structure have to be extracted.

Analogously, the presence of a grouping node m in the query graph specified in a constructing rule implies that the subtree rooted in m must be replicated for all the possible instances of the variables labelling the node in the subtree.

The formal semantics of the construction rule is given expressing, by means of logical rules with complex terms and nested sets, the structure of the output graph. Complex terms and nested sets are used to describe the nesting of elements and grouping elements. (An informal semantics of logic languages with sets is provided in the Appendix; for the formal semantics we address readers to [7].) Before introducing the semantics of construction rules we present an example.

Example 9. Consider the second query of Example 1. The logic rule defining the output graph is the following:

$$\begin{array}{l}\texttt{Tree(xml(results(}\bot, \ll\texttt{result(}\bot\texttt{, title(T), author(A))}\gg\texttt{)))}\\ \quad\leftarrow \texttt{Theta(T, A)}\end{array}$$

where $\bot$ is the null value denoting that the element `<results>` does not have attributes and only contains a set of subelements `<result>`. The condition Θ is defined by the following rule:

$$\begin{array}{l}\texttt{Theta(T, A)} \leftarrow \texttt{Node(Id}_\texttt{A}\texttt{, A, \$a)} \wedge \texttt{Node(Id}_\texttt{T}\texttt{, T, \$t)}\\ \qquad\qquad\qquad \wedge\, \texttt{arc(X, Id}_\texttt{A}\texttt{, <author>)} \wedge \texttt{arc(X, Id}_\texttt{T}\texttt{, <title>)}\end{array}$$

Analogously, the logic rule defining the output graph for the fourth query of Example 1 is as follows:

$$\begin{array}{l}\texttt{Tree(xml(results(}\bot, \ll\texttt{result(}\bot, \ll\texttt{title(T)}\gg\texttt{, author(A))}\gg\texttt{)))}\\ \quad\leftarrow \texttt{Theta(T, A)}\end{array}$$

The logic program associated to a rewritten $\mathcal{XGL}$ constructing rule (α', Θ') consists of a logical program defining two predicates *Tree* and *Theta*. The predicate *Tree* defines the structure of the resulting document, whereas *Theta* is the translation of the *FO* formula Θ' into a logic program.

The predicate *Tree* is defined by a rule of the form $\texttt{Tree(xml(TR(}\alpha'\texttt{))} \leftarrow \texttt{Theta(V}_1\texttt{, } \ldots\texttt{, V}_\texttt{n}\texttt{)}$, where $V_1, \ldots, V_n$ are the variables appearing in α, and $\texttt{TR(}\alpha'\texttt{)}$ denotes the translation of α' into a nested structure defined as follows:

- if α' is a leaf node with label n, $\texttt{TR(}\alpha'\texttt{)}$ denotes the label of n;
- if the root of α' is a nonleaf node n, $T_1, \ldots, T_q$ are the subtrees connected to n whose root is not a grouping node and $T'_1, \ldots, T'_r$ are the subtrees connected to n whose root is a grouping node, then $\texttt{TR(}\alpha'\texttt{)}$ denotes: $\texttt{X, l}_1\texttt{(TR(T}_1\texttt{))}, \ldots, \texttt{l}_\texttt{q}\texttt{(TR(T}_\texttt{q}\texttt{))}$, $\ll \texttt{l}'_1\texttt{(TR(T}'_1\texttt{))} \gg, \ldots, \ll \texttt{l}'_\texttt{r}\texttt{(TR(T}'_\texttt{r}\texttt{))} \gg$ where X is the label of $root(\alpha')$ and $l_1, \ldots, l_q, l'_1, \ldots, l'_r$ are the labels of the arcs connecting $T_1, \ldots, T_q, T'_1, \ldots, T'_r$ to n.

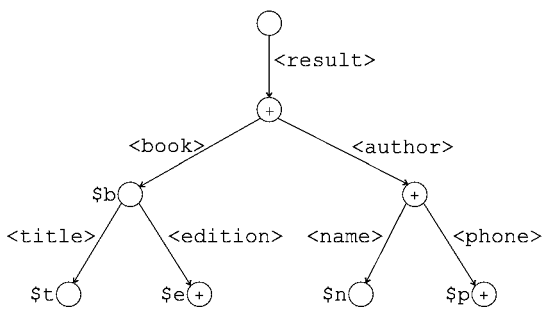

Figure 18. Graph of a construction rule.

Observe that if a node of α' is labelled with a variable V, V also appears as an argument of the atom $\mathit{Theta}(V_1, \ldots, V_n)$, even if it does not appear in the original condition Θ.

Example 10. Consider the constructing rule of Figure 18 where the condition Θ has not been reported. The associated logic rule is as follows:

$$\mathtt{Tree(xml(}\bot, \ll\mathtt{result(}\bot, \mathtt{book(B, title(T)}, \ll\mathtt{edition(E)}\gg),$$
$$\ll\mathtt{author(}\bot, \mathtt{name(N)}, \ll\mathtt{phone(P)}\gg)\gg\gg)) \leftarrow \mathtt{Theta(B, T, E, N, P)}$$

Observe that the first-order formula specified in the constructing rule has a different purpose from that specified in the extracting rule: the former consists of join predicates (for associating the extracted data to the nodes of the graph corresponding to the document to be constructed), the latter of selection predicates for filtering information from the source documents.

The following theorem characterizes the complexity of answering to an $\mathcal{XGL}$ query.

Proposition 1. Let Q be an $\mathcal{XGL}$ query and $D_1, \ldots, D_k$ a set of XML documents. The computation of a document in $Q(D_1, \ldots, D_k)$, selected nondeterministically, can be done in polynomial time (w.r.t. the size of the documents).

Proof: The computation of a document in $Q(D_1, \ldots, D_k)$ consists of three steps: (i) extracting k subgraphs from the graphs corresponding to the documents $D_1, \ldots, D_k$ (according to the extracting rules of the `WHERE` clause of Q), (ii) renaming variables, and (iii) constructing a new document according to the `CONSTRUCT` clause. From Theorem 1, we have that the first step can be done in polynomial time, since it corresponds to finding k minimal mapping pairs (one for each data graph corresponding to the documents $D_1, \ldots, D_k$). The second step can be done in constant time, since we can assume that the number of variables used in the extracting rules is constant w.r.t. the size of the specified documents. The third step corresponds to the evaluation of a datalog rule, that is feasible in polynomial time [4]. $\square$

6.3. $\mathcal{XGL}$ versus other XML query languages

In this section we rely on some examples to give a flavor of the differences and similarities among our language and other proposals (XQuery, XML-QL and XML-GL). With respect to other declarative languages, $\mathcal{XGL}$ seems to be more intuitive and natural. Indeed, as shown by the following example, in some cases textual languages are not easy to use for expressing queries over graph-like data.

Example 11. The fourth query of Example 1 (constructing a document which contains for each author the titles of the books he has written) can be expressed in XML-QL as follows:

```
CONSTRUCT <results> {
   WHERE
      <bib>
         <author> $a </author>
      </bib>
   CONSTRUCT
      <result>
         <author> $a </author>
         {
            WHERE
               <bib>
                  <book>
                     <title> $t </title>
                     <author> $a </author>
                  </book>
               </bib> IN "bib.xml"
               CONSTRUCT <title> $t </title>
         }
      </result>
} </results>
```

The same query in XQuery is taken from [37] and is shown below:

```
FOR $a IN distinct-value($bib/book/author/data())
RETURN
      <biblio>
        <author>{ $a }</author>
        { FOR $b IN $bib/book,
                $a2 IN $b/author/data()
          WHERE $a = $a2
          RETURN $b/title
        }
      </biblio>
```

The same query can be expressed in $\mathcal{XGL}$ using the parsing grammar and the constructing rule reported, respectively, on the left side and on the right side of Figure 19:

```
WHERE S IN "bib.xml"
CONSTRUCT T
```

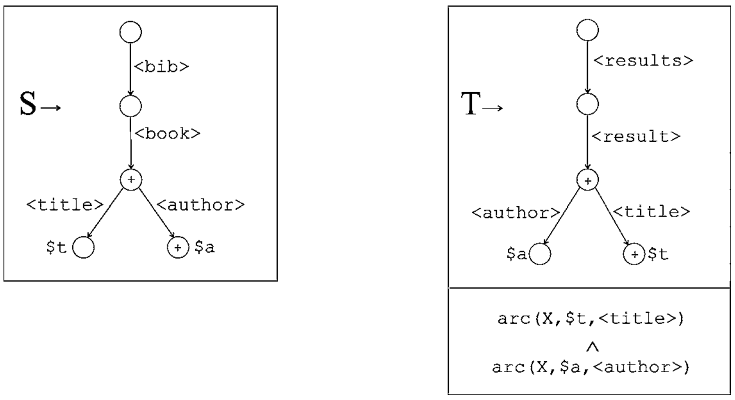

Figure 19. Fourth query of Example 1.

Example 12. Suppose we want to display, for each publisher, the authors and the titles of the books edited by the same publisher. This query can be expressed in XQuery as follows:

```
FOR $p IN document(bib.xml)//publisher
RETURN
  <result>
    <publisher> $p/text() </publisher>
    {
      FOR $b IN document(bib.xml)//book[/publisher=$p]
      RETURN $b/title
    }
    { FOR $a IN distinct-value(document(bib.xml)//
                              author[$p=/../publisher]/data())
      RETURN $a
    }
  </result>
```

The above query is quite complex, since an author may have written more than one book with the same publisher and so we have to avoid that the list of authors contains duplicates.
We can write the same query in $\mathcal{XGL}$ (S and T are those of Figure 20):

```
WHERE S IN "bib.xml"
CONSTRUCT T
```

With respect to XML-GL, our graphical notation is based on the use of graph grammar production rules. In the following example we define a query and express it both in XML-GL and $\mathcal{XGL}$.

Example 13. We consider here a query taken from [11]. We are given an XML document `addrbook.xml` containing a collection of people where each person has an address and

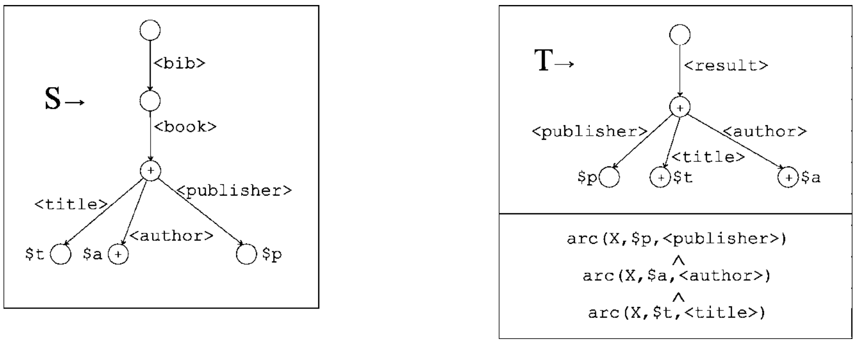

Figure 20. The query of Example 12 written in $\mathcal{XGL}$.

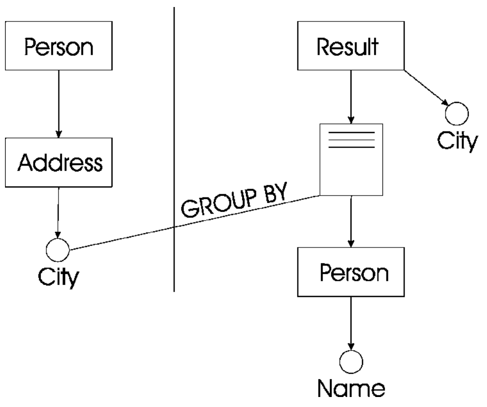

Figure 21. A query in XML-GL.

inside the address the city is specified. The problem consists of constructing a document which contains, for each city, the people living in the city.

The query can be expressed in XML-GL as shown in Figure 21, and in $\mathcal{XGL}$ as shown in Figure 22:

```
WHERE S IN "addrbook.xml"
CONSTRUCT T
```

7. Conclusions

In this paper we have presented a graphical language for XML data. We have introduced parsing graph grammars and constructing rules which allow users to query and construct graph-like data in an easy, powerful and flexible way. Moreover, since most of the recursive queries on graph-like databases are of the form "find all nodes reachable from a given node" (path queries) or find the list of nodes which are directly (or indirectly) connected to a given node, we have introduced shortcuts which allow users to write, in most cases,

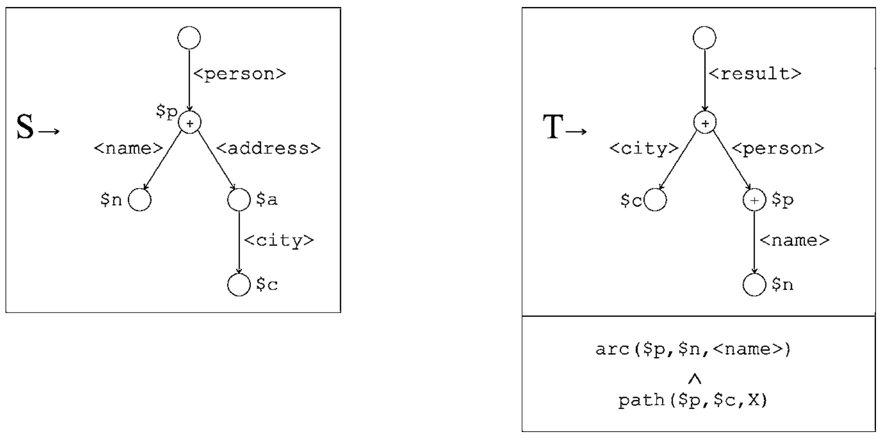

Figure 22. The same query of Figure 21 in $\mathcal{XGL}$.

(recursive) queries "without recursion" by means of a very limited number of productions (all examples presented here use only one production).

The $\mathcal{XGL}$ language is currently under implementation: its features have been improved by introducing aggregates and negation; the semantics has been extended to deal with ordered XML graphs.

Appendix A. Sets in logic languages

Several proposals for handling sets in logic languages have been made in the literature [3,7, 29]. We refer to the proposal of [7] that is particularly relevant not only from a theoretical point of view, but also because it has been practically used in many applications.

A simple term is either a constant or a variable. A term can be either a simple term, a complex term or a set term. A complex term is of the form $f(t_1, \ldots, t_n)$ where f is a term constructor (i.e. a function symbol not used recursively) and t_j $(1 \leqslant j \leqslant n)$ is a term. A *set term* S is a term of the form $\{s_1, \ldots, s_n\}$, where s_j $(1 \leqslant j \leqslant n)$ is a term and the sequence in which the elements are listed is not immaterial.

We point out that the enumeration of the elements of a set term can be given either directly or by giving the conditions for collecting their elements (*grouping variables*). Grouping variables may occur in the head of clauses with the following format

$$p(x_1, \ldots, x_h, \ll Y \gg) \leftarrow B_1, \ldots, B_n,$$

where $B_1, \ldots, B_n$ are the goals of the rules, p is the head predicate symbol with parity $h + 1$, $\langle Y \rangle$ is a grouping variable, and $x_1, \ldots, x_h$ are the other arguments (terms or other grouping variables). The term $\langle Y \rangle$ will be eventually assigned the set $\{Y\theta \mid \theta$ is a substitution for r such that $B_1\theta, \ldots, B_n\theta$ are true$\}$. Thus, a grouping variable is similar to the construct *GROUP BY* of SQL or the built-in predicate *setof* of PROLOG.

Example 14. Consider the *Supply* relation below:

Supplier	Part	Quantity
s1	p1	10
s1	p1	20
s1	p2	10
s1	p2	30
s2	p1	5
s2	p1	8

The following rule:

$$\texttt{part\_set(S,}\ll\texttt{P}\gg\texttt{)} \leftarrow \texttt{supplier(S,P,Qty)}$$

collects all the parts supplied by the suppliers `s1` and `s2` in two sets, i.e., `part_set(s1,{p1,p2})` and `part_set(s2,{p1})`. The rule

$$\texttt{part\_set(S,}\ll\texttt{part(P,}\ll\texttt{Qty}\gg\texttt{)}\gg\texttt{)} \leftarrow \texttt{supplier(S,P,Qty)}$$

computes the tuples
```
  part_set(s1, { part(p1,{10,20}), part(p2, {10,30}) })
```
and
```
  part_set(s2, { part(p1,{5,8}) }).
```

Appendix B. Further examples

Example 15. Consider the use case in [38, Section 1.9]. Suppose that the file `census.xml` contains an element `<person>` for each person recorded in a recent census. For each person element, the person's name, job, and spouse (if any) are recorded as attributes. The `spouse` attribute is an IDREF-type attribute that matches the ID-type `name` attribute of the spouse element.

The parent–child relationship among persons is recorded by containment in the element hierarchy: the element that represents a child is contained within the element that represents the child's father or mother. A child is recorded under either its father or its mother (but not both): in the following, the term "*children of X*" includes "*children of the spouse of X*." Each person in the census has zero, one, or two parents. An input document `census.xml` and its DTD are shown below:

```
<census>
  <person name="Bill" job="Teacher">
    <person name="Joe" job="Painter" spouse="Martha">
      <person name="Sam" job="Nurse">
        <person name="Fred" job="Senator" spouse="Jane">
        </person>
      </person>
      <person name="Karen" job="Doctor" spouse="Steve">
      </person>
    </person>
    <person name="Mary" job="Pilot">
      <person name="Susan" job="Pilot" spouse="Dave">
      </person>
    </person>
  </person>
  <person name="Frank" job="Writer">
    <person name="Martha" job="Programmer" spouse="Joe">
      <person name="Dave" job="Athlete" spouse="Susan">
      </person>
    </person>
    <person name="John" job="Artist">
      <person name="Helen" job="Athlete">
      </person>
      <person name="Steve" job="Accountant" spouse="Karen">
        <person name="Jane" job="Doctor" spouse="Fred">
        </person>
      </person>
    </person>
  </person>
</census>

<!DOCTYPE census [
  <!ELEMENT census (person*)>
  <!ELEMENT person (person*)>
  <!ATTLIST person
        name    ID      #REQUIRED
        spouse  IDREF   #IMPLIED
        job     CDATA   #IMPLIED >
]>
```

Now consider query Q5 of Section 1.9 in [38]: list the names of parents and children who have the same jobs, and their jobs. Here it is the query in XQuery:

```
<result>
{
    FOR $p IN document("census.xml")//person,
        $c IN $p/person[job = $p/job]
    RETURN
       <match parent={ $p/name } child={ $c/name }
              job={ $c/job } />
}
```

```
{
    FOR $p IN document("census.xml")//person,
        $c IN $p/@spouse->person/person[job = $p/job]
    RETURN
       <match parent={ $p/name } child={ $c/name }
              job={ $c/job } />
}
</result>
```

The same query in $\mathcal{XGL}$ is the following:

```
WHERE S IN "census.xml"
CONSTRUCT T [$p->$p1,$p->$p2]
```

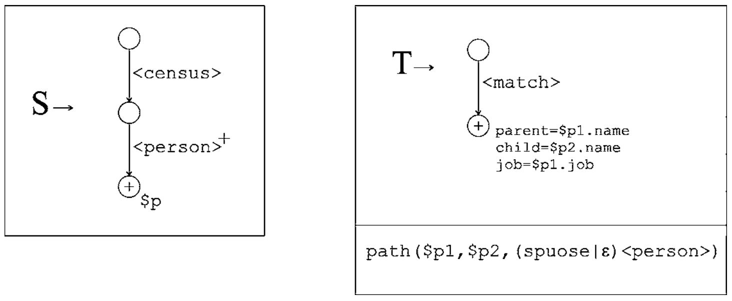

Figure 23. Parents and children with the same job.

Example 16. We want to extract the list of name-pairs of grand-parents and grandchildren from the document census.xml of the previous example (see query Q7 in [38, Section 1.9]. The query can be expressed in XQuery as follows:

```
<results>
  {
    FOR $b IN document("census.xml")//person,
        $c IN $b/person | $b/@spouse->person/person,
        $g IN $c/person | $c/@spouse->person/person
    RETURN
        <grandparent name={ $b/name }
                     grandchild={ $g/name } />
  }
</results>
```

The same query can be written in $\mathcal{XGL}$ as follows:

```
WHERE S IN "census.xml"
CONSTRUCT T [$p->$p1,$p->$p2]
```

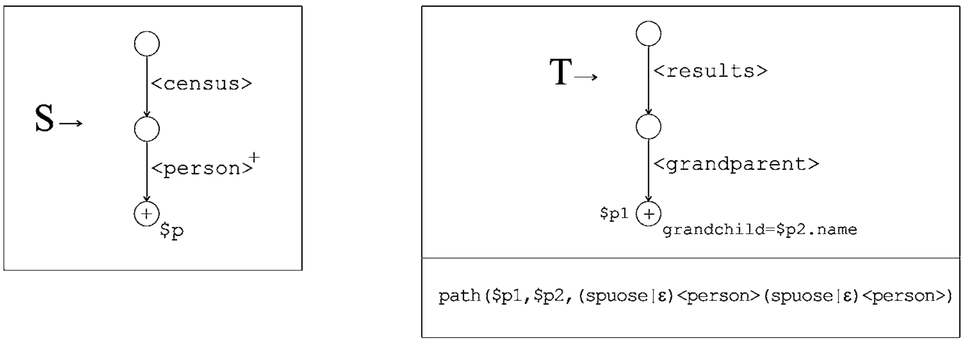

Figure 24. Pairs grandparent–grandchild.

```
<standing>
  <team> Reds
    <player> John </player>
    <player> Bill </player>
    <player> Cathy </player>
  </team>
  <team> Blacks
    <player> Peter </player>
    <player> Jean </player>
  </team>
  <team> Clouds
    <player> Sue </player>
    <player> Mark </player>
    <player> Ed </player>
  </team>
</standing>
```

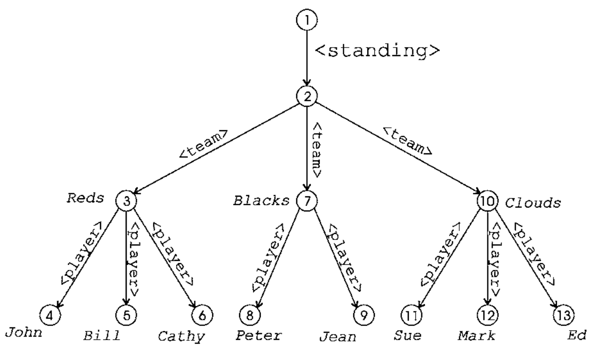

Figure 25. An XML document and the corresponding (ordered) XML graph.

Appendix C. Introducing order

In this section we introduce the possibility of querying and creating ordered XML documents by means of $\mathcal{XGL}$. Such a feature is significant, since the order among elements in a document is generally an important information.

In the previous sections we disregarded the order of the input and output documents for the sake of simplicity. Here we informally introduce the features of the language related to the management of order.

We assume that the arcs (and the nodes) of an XML graph are ordered according to the structure of the corresponding document. That is, given two elements e_1, e_2, $e_1 > e_2$ iff the starting tag of e_1 appears before the starting tag of e_2 in the document.

For instance, the XML document `census.xml` on the left side of Figure 25 corresponds to the ordered graph on the right side.

The number inside each node represents the position of the corresponding element in the source document.

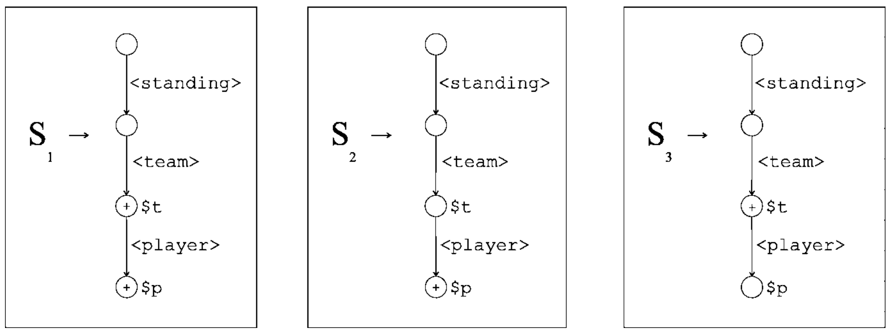

Figure 26. Three $\mathcal{XGL}$ parsing grammars.

S_1	S_2	S_3
`Node(3,'Reds',$t)`	`Node(3,'Reds',$t)`	`Node(3,'Reds',$t)`
`Node(4,'John',$p)`	`Node(4,'John',$p)`	`Node(4,'John',$p)`
`Node(5,'Bill',$p)`	`Node(5,'Bill',$p)`	`Node(7,'Blacks',$t)`
`Node(6,'Cathy',$p)`	`Node(6,'Cathy',$p)`	`Node(8,'Peter',$p)`
`Node(7,'Blacks',$t)`		`Node(10,'Clouds',$t)`
`Node(8,'Peter',$p)`		`Node(11,'Sue',$p)`
`Node(9,'Jean',$p)`		
`Node(10,'Clouds',$t)`		
`Node(11,'Sue',$p)`		
`Node(12,'Mark',$p)`		
`Node(13,'Ed',$p)`		

Figure 27. Lists of tuples extracted by S_1, S_2 and S_3.

As to the extraction of information from an ordered XML graph, we point out that the definition of data mapping can be trivially extended to deal with ordered documents, and for this reason will be not discussed here. We only observe that the main consequence of the introduction of order is that the tuples in the relation *Node* are ordered w.r.t. the position of the corresponding nodes.

For instance, consider the results of the extraction phases performed, respectively, by applying the following $\mathcal{XGL}$ parsing grammars on the XML document shown in Figure 25.

The grammar S_1 extracts all the teams and all the players from the document, S_2 extracts the first team and its players, whereas S_3 extracts all the teams and, for each team, the first of its players. The ordered sets of tuples corresponding to the terminal mapping pairs obtained at the end of the three extraction processes are represented in the columns of the table in Figure 27.

The order on the set of tuples corresponding to the terminal mapping pair obtained at the end of the extraction process induces a partial order in the set of tuples satisfying Θ. We point out that the constructing rule returns an ordered *Tree*, i.e. a grouping variable

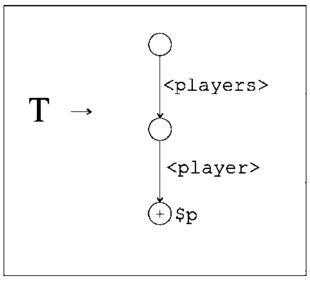

Figure 28. An $\mathcal{XGL}$ constructing rule.

S_1	S_2	S_3
`<players>`	`<players>`	`<players>`
`<player> John </player>`	`<player> John </player>`	`<player> John </player>`
`<player> Bill </player>`	`<player> Bill </player>`	`<player> Peter </player>`
`<player> Cathy </player>`	`<player> Cathy </player>`	`<player> Sue </player>`
`<player> Peter </player>`	`</players>`	`</players>`
`<player> Jean </player>`		
`<player> Sue </player>`		
`<player> Mark </player>`		
`<player> Ed </player>`		
`</players>`		

Figure 29. Document created by T after applying S_1, S_2 and S_3.

corresponds to a list instead of a set. Such a list is ordered according to the partial order defined on Θ.

For instance, consider the $\mathcal{XGL}$ constructing rule T in Figure 28, which returns the ordered list of the players generated in the extraction phase. The table in Figure 29 shows the generated documents when T is applied, respectively, after S_1,S_2 and S_3.

Note that the constructing rule T of Figure 28 in all of the three cases returns the complete sequence of the extracted players in the same order as they appear in the document.

However, in many cases we may want to change the order of the elements w.r.t. the source document, or return only some elements depending on their position. To this aim, we have introduced the clauses `ORDER BY` and `RANGE`.

The clause `ORDER BY` takes three arguments:

(1) the root of subtree which has to be reordered;
(2) the list of the nodes contained in such a subtree whose values determine the order of the elements in the generated graph;
(3) the ordering direction (ascending or descending).

The clause `RANGE` takes two arguments:

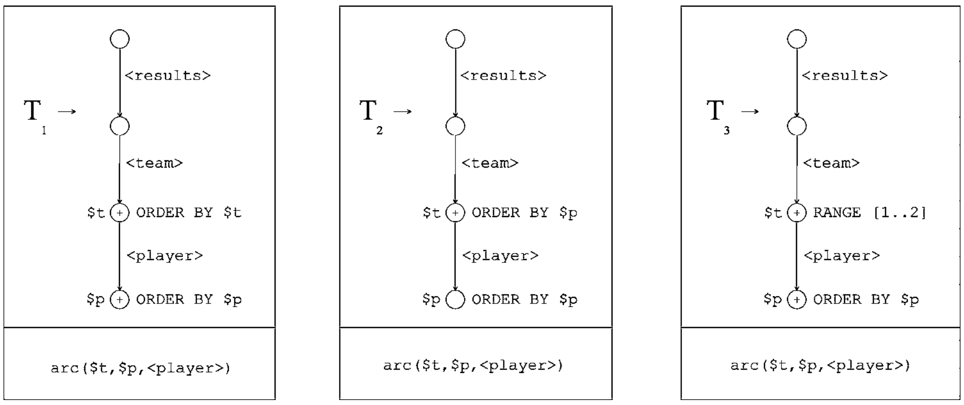

Figure 30. Three $\mathcal{XGL}$ constructing rules.

T_1	T_2	T_3
<results>	<results>	<results>
<team> Blacks	<team> Reds	<team> Reds
<player> Jean </player>	<player> Bill </player>	<player> Bill </player>
<player> Peter </player>	</team>	<player> Cathy </player>
</team>	<team> Clouds	<player> John </player>
<team> Clouds	<player> Ed </player>	</team>
<player> Ed </player>	</team>	<team> Blacks
<player> Mark </player>	<team> Blacks	<player> Jean </player>
<player> Sue </player>	<player> Jean </player>	<player> Peter </player>
</team>	</team>	</team>
<team> Reds	</results>	</results>
<player> Bill </player>		
<player> Cathy </player>		
<player> John </player>		
</team>		
</results>		

Figure 31. Document created by T_1, T_2 and T_3 after applying S_1.

(1) a node;
(2) the range defining the position in the source document of the elements which can be associated to the specified node.

Consider, for instance, the three constructing rules in Figure 30. Rule T_1 returns all the extracted teams ordered by their name and, for each team, the list of its players ordered by their name. Rule T_2 returns all the extracted teams ordered by the player of theirs which has the "minimum" name: that is, the team `Clouds` precedes the team `Blacks` since the first `Clouds`'s player (`Ed`) precedes lexicographically the first of `Blacks`'s player (`Jean`). Finally, rule T_3 returns the teams which appear in the first two positions in the document, and for each team the list of players ordered by their names.

Table in Figure 31 reports the documents obtained by extracting information by means of S_1 and then restructuring the data by means of T_1, T_2 and T_3.

Notes

1. The mapping ψ replaces variables in α with constants in the input data graph.
2. A this level we use the same syntax of XML-QL [18].
3. In the prototype of the language under development we use FO_R^{agg} formulas. FO_R^{agg} denotes first order logic over the signature of the real field with aggregation operators [26].
4. Formally, since variables in the extracted graph may be renamed, we should use an additional predicate storing the mapping between variables in the extracted graph and (renamed) variables in the output graph.

References

[1] S. Abiteboul, "Semistructured data," in *Proc. International Conference on Database Theory*, Delphi, Greece, 1997, pp. 1–18.

[2] S. Abiteboul, P. Buneman, and D. Suciu, *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kauffman, San Francisco, CA, 1999.

[3] S. Abiteboul and S. Grumbach, "COL: A logic-based language for complex objects," in *Proc. Int. Conf. on Extending Database Technology*, Venice, Italy, 1988, pp. 271–293.

[4] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison-Wesley, Boston, MA, 1994.

[5] S. Abiteboul and V. Vianu, "Regular path queries with constraints," in *Proc. PODS*, Tucson, Arizona, 1997, pp. 122–133.

[6] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener, "The Lorel query language for semistructured data," *Journal of Digital Libraries* 1(1), 1997, 68–88.

[7] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur, "Set constructors in a logic database language," *Journal of Logic Programming* 10(1/2/3&4), 1991, 181–232.

[8] A. Bonifati and S. Ceri, "Comparative analysis of five XML query languages," *SIGMOD Record* 29(1), 2000, 68–79.

[9] P. Buneman,"Semistructured data," in *Proc. PODS*, Tucson, AZ, 1997.

[10] P. Buneman, W. Fan, and S. Weinstein, "Path constraints in semistructured and structured databases," in *Proc. PODS*, Austin, TX, 1988, pp. 129–138.

[11] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca, "XML-GL: A graphical language for querying and restructuring XML documents," *Computer Networks* 31(11–16), 1999, 1171–1187.

[12] S. Ceri, S. Comai, E. Damiani, P. Fraternali, and L. Tanca, "Complex queries in XML-GL," in *ACM Symp. on Applied Computing*, Vol. 2, 2000, pp. 888–893.

[13] S. Ceri, P. Fraternali, and S. Paraboschi, "XML: Current developments and future challenges for the database community," in *Proc. Int. Conf. on Extending Database Technology*, Como, Italy, 2000, pp. 3–17.

[14] V. Christophides, S. Cluet, and G. Moerkotte, "Evaluating queries with generalized path expressions," in *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, 1996, pp. 413–422.

[15] M. Consens and A. Mendelzon, "GraphLog: A visual formalism for real life recursion," in *Proc. PODS*, Nashville, TN, 1990, pp. 404–416.

[16] E. Damiani and L. Tanca, "Blind queries to XML data," in *Proc. Int. Conf. on Database and Expert Systems Applications*, London/Greenwich, UK, 2000, pp. 345–356.

[17] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu, "XQuery: A query language for XML," 2000, `http://www.w3.org/TR/2001/WD-xquery-20010215`

[18] A. Deutsch, M. F. Fernandez, D. Florescu, D. A. Levy, and D. Suciu, "A query language for XML," *Computer Networks* 31(11–16), 1999, 1155–1169.

[19] J. Engelfriet, "Context-free graph grammars," in *Handbook of Formal Languages*, Vol. 3, Beyond Words, G. Rozenberg and A. Salomaa, Eds., Springer-Verlag, London, UK, 1997, pp. 125–213.

[20] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu, "STRUDEL: A web-site management system," in *Proc. ACM SIGMOD Conf. on Management of Data*, Tucson, AZ, 1997, pp. 549–552.

[21] M. F. Fernandez, J. Simeon, and P. Wadler, "An algebra for XML query," in *Int. Conf. on Found. of Software Technology and Theoretical Computer Science*, New Delhi, India, 2000, pp. 11–45.

[22] S. Flesca, F. Furfaro, and S. Greco, "Graph grammars for querying graph-like data," in *Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT)*, Crete, Greece, 2001.

[23] S. Flesca, F. Furfaro, and S. Greco, "A graph grammars based framework for querying graph-like data," ISI-CNR Technical Report 8, Cosenza, Italy, 2002, submitted for publication.

[24] S. Flesca and S. Greco, "Querying graph databases," in *Proc. Int. Conf. on Extending Database Technology*, Konstanz, Germany, 2000, pp. 510–524.

[25] S. Flesca and S. Greco, "Partially ordered regular languages for graph queries," in *Proc. Int. Colloquium on Automata, Languages and Programming*, Prague, Czech Republic, 1999, pp. 321–330.

[26] S. Grumbach, M. Rafanelli, and L. Tininini, "Querying Aggregate Data," in *Proc. PODS*, Philadephia, PA, 1999, pp. 174–184.

[27] Z. G. Ives and Y. Lu, "XML query languages in practice: An evaluation," in *Proc. Web-Age Information Management*, Shanghai, China, 2000, pp. 29–40.

[28] D. Konopnicki and O. Shmueli, "W3QS: A query system for the World-Wide-Web," in *Proc. Int. Conf. on Very Large Data Bases*, Zürich, Switzerland, 1995, pp. 54–65.

[29] G. M. Kuper, "Logic programming with sets," *Journal of Computer and System Science* 41, 1990, 44–64.

[30] L. Lakshmanan, F. Sadri, and I. Subramanian, "A declarative language for querying and restructuring the web," in *Proc. Int. Workshop on Research Issues in Data Engineering*, New Orleans, LA, 1996, pp. 12–21.

[31] G. Mecca, P. Atzeni, A. Masci, P. Merialdo, and G. Sindoni, "The Araneus web-base management system," in *Proc. of SIGMOD Conference*, Seattle, WA, 1998, pp. 544–546.

[32] A. Mendelzon, G. Mihaila, and T. Milo, "Querying the World Wide Web," *Journal of Digital Libraries* 1(1), 1997, 54–67.

[33] B. Oliboni and L. Tanca, "Querying XML specified WWW sites: Links and recursion in XML-GL," in *Proc. Int. Conf. Computational Logic*, London, UK, 2000, pp. 1167–1181.

[34] J. Paredaens, P. Peelman, and L. Tanca, "G-Log: A declarative graphical query language," in *Proc. Int. Conf. on Deductive and Object-Oriented Databases*, Munich, Germany, 1991, pp. 108–128.

[35] D. Suciu, "Semistructured Data and XML," in *Proc. Int. Conf. on Foundations of Data Organization and Algorithms*, Kobe, Japan, 1998.

[36] XML query requirements, W3C working draft, `http://www.w3.org/TR/xmlquery-req`

[37] XQuery formal semantics, W3C working draft, `http://www.w3.org/TR/query-semantics`

[38] XQuery use cases, W3C working draft, `http://www.w3.org/TR/xmlquery-use-cases`