XML-GL: a Graphical Language for Querying and Restructuring XML Documents

S. Ceri, S. Comai, E. Damiani[†], P. Fraternali, S. Paraboschi, L. Tanca

Email: ceri,comai,fraterna,parabosc,tanca@elet.polimi.it, edamiani@crema.unimi.it

Politecnico di Milano, Dipartimento di Elettronica e Informazione

† Università di Milano, Polo di Crema

Abstract

The widespreading of XML as a standard for semi-structured documents on the WEB opens up challenging opportunities for WEB query languages. In this paper we introduce XML-GL, a graphical query language for XML documents. The use of a visual formalism for representing both the content of XML documents (and of their DTDs) and the syntax and semantics of queries enables an intuitive expression of queries, even when they are rather complex. XML-GL is inspired by G-log, a general purpose, logic-based language for querying structured and semi-structured data. The paper presents the basic capabilities of XML-GL through a sequence of examples of increasing complexity and discusses advanced query primitives like order-sensitive conditions, arithmetics and aggregate functions, and complex document construction.

1 Introduction and Motivations

XML [Con98] is a recent recommendation of the World Wide Web Consortium for a meta-language to define markups for content publishing on the Web. The design goals of XML are driven by the half-a-decade experience of usage of HTML as a content description language, which has exposed several inadequacies:

- The HTML tag set is fixed, and its extension to cover new application requirements either breaks the standard or demands a long standardization process.
- The HTML mark-up intermixes structural and visual annotations, producing documents which are hard to process by software agents searching for information on the Web.

XML addresses both problems by letting content producers define and use the set of tags that best mirrors the structure and conceptual properties of the content they want to publish.

The shift from HTML to XML brings a major change in the structure of Web information, which becomes more and more a collection of semi-structured objects, i.e., pieces of content for which at least a partial representation of structure (known as *schema*) is available. This evolution brings forth the necessity of novel languages for extracting information from XML content, much in the same way as traditional query languages (notably SQL) have been used for extracting information from structured data, e.g., relational database.

As in database applications, the purpose of a query language is twofold:

- Letting users extract information from data repositories;
- Restructuring information stored in one or more repositories to match novel users' needs.

XML-GL addresses both issues, by permitting the formulation of queries for extracting information from XML documents and for restructuring such information into novel XML documents.

The originality of XML-GL with respect to other proposals for querying XML documents, like XML-QL [DFF+98], XQuery [DeR98], XQL [IKK98, RLS98], and XSSL [W3C98], is that queries are formulated visually, using a graph-based formalism close to the structure of XML documents (e.g., comparable to the visual representations of XML documents offered by XML authoring tools like the Near & Far XML editor). However, XML-GL is not a visual interface over a conventional, textual, query language, but a graph-based query language with both its syntax and semantics defined in terms of graph structures and operations [PPT95, SERL98, CDQT98].

1.1 Requirements for an XML Query Language

Prior to introducing the features of XML-GL, we propose a set of requirements for the design of a query language over XML documents.

1. The language should be flexible enough to allow *query formulation both for valid and well-formed documents*. Availability of a DTD should result in a facilitation of both expressing and evaluating a query in XML-GL.
2. We want to address *queries to a whole WEB site*, taking into account links between different documents, rather than querying only one document at a time.
3. We want the possibility to *access DTDs* and *XML-based metadata* (such as those proposed in the RDF draft standard proposal [BGL98]) as well as data, by using the same query paradigm.
4. We want to *extract* information from an XML document, by means of powerful, yet declarative, pattern matching and element manipulation primitives.
5. We want to *reshape* the source XML documents, by specifying new links between existing elements as well as new elements in the XML document resulting from a query. This last process requires the introduction of new tags in the result document.
6. We want to specify *regular expressions on paths*, i.e. the possibility of following recursively arbitrarily long paths in a site, possibly specifying conditions on path nodes. This feature is particularly helpful in the case the search is directed towards certain document patterns, that may be placed arbitrarily in the XML source.
7. We also want to support arbitrary computations on the numeric content of documents by means of built-in functions.
8. Given that XML is positional, we may want to allow an *order-sensitive query interpretation*, i.e. one in which the relative order of appearance of XML tags and character data is meaningful. However, such an interpretation may be overly restrictive, so the query language should also offer an unordered interpretation (probably as default).
9. Under a given query interpretation, XML-GL should be able to compute approximate results, *similar* to the ones that fully satisfy the queries.

10. Finally, we want queries in our language to be *readable*, and the DTD of the result of the query immediately apparent.

2 The XML-GL Data Model

Query languages for semi-structured and strucured data rely on *data models*, which are abstract notations for representing the organization of data. Data models are used to define *data schemas*, i.e., actual data organizations. The difference between structured and semi-structured data is that the former, e.g., tuples in a relational database, always obey an explicit data schema, whereas the latter may have only a partial schema or no schema at all (e.g., an HTML page has always a head and a body, but the content of both may present different structure from page to page).

XML permits the representation of both structured and semi-structured data. On one hand, it provides a notion similar to that of a data schema to express the organization of documents: an XML document can be compliant to a Document Type Definition (DTD), that specifies the types of markup elements that can appear in the document, their attributes and containment relationships. If an XML document adheres to a DTD, it is said to be *valid*. On the other hand, XML documents need not be valid; if they lack a DTD but respect some syntactic rules for tag placement they are said to be *well formed*.

XML could be directly assumed as the data model for a query language. However, it includes also document-oriented features different from those found in classical (database-style) data models, which can be safely abstracted away in the definition of a query language. For coherence with the visual nature of XML-GL, we introduce an explicit data model for XML documents, called XML-GDM (XML Graphical Data Model), which we use to represent both the expected structure of XML documents (i.e., their DTDs) and actual documents. XML-GDM syntax will be used also (with a few additional graphic notations) for writing XML-GL queries and for representing the DTD of their result, with the benefit of reducing to the minimum the notations that the user should learn to query XML documents.

Well-formed documents without a DTD can be queried in the same way as valid documents: however, since no information about the structure of the document is available at query formulation time, it is more likely that queries may not match exactly the structure of the document and result in empty answers. Some flexibility in the definition of the data model and query language may alleviate the problem of querying documents without a DTD, by allowing syntactically different, but semantically equivalent, representations of the same information and leveraging data and query transformation techniques to improve the response to imprecise queries.

To present XML-GDM, we introduce an example of a DTD and an XML document conforming to it.

2.1 Running Example

Consider the DTD shown in Figure 1, which specifies the structure of documents containing book orders. For each order the information about the consignee, the ordered items, the date, and possibly a contact address are given. Each item contains the information about the book, the quantity and possibly the discount percentage. For each book the ISBN code, the price, and possibly the title and the authors are known. A possible XML document conforming to this DTD is shown in Figure 2.

```
<!ELEMENT order (shipto, contact?, item+, date)>
<!ATTLIST order number PCDATA #REQUIRED>
<!ELEMENT shipto (fulladdress|reference)>
<!ELEMENT contact (reference|PCDATA)>
<!ELEMENT fulladdress (company?, city, addressline+)>
<!ELEMENT reference EMPTY>
<!ATTLIST reference customer IDREF>
<!ELEMENT person (firstname?,lastname,fulladdress)>
<!ATTLIST person id ID>
<!ELEMENT company PCDATA>
<!ELEMENT addressline PCDATA>
<!ELEMENT city PCDATA>
<!ELEMENT date (day, month, year)>
<!ELEMENT day PCDATA>
<!ELEMENT month PCDATA>
<!ELEMENT year PCDATA>
<!ELEMENT item (book, quantity, discount?)>
<!ELEMENT book (isbn,title?,price,author*)>
<!ELEMENT author (firstname?,lastname)>
<!ELEMENT firstname PCDATA>
<!ELEMENT lastname PCDATA>
<!ELEMENT isbn PCDATA>
<!ELEMENT title PCDATA>
<!ELEMENT price PCDATA>
<!ELEMENT quantity PCDATA>
<!ELEMENT discount PCDATA>
```

Figure 1: DTD of the Running Example

```
<?xml version="1.0" standalone="no" encoding="UTF-8"?>
<DOCTYPE ORDER SYSTEM "order.dtd">

<ORDER number=1>
 <SHIPTO><REFERENCE customer="C00001"></REFERENCE></SHIPTO>
 <CONTACT>Tim Bell</CONTACT>ITEM>
 <DATE><DAY>14</DAY><MONTH>11</MONTH><YEAR>1998</YEAR></DATE>
 <ITEM>
  <BOOK><ISBN>15536455</ISBN>
    <TITLE>Introduction to XML</TITLE>
    <PRICE>24.95</PRICE>
    <AUTHOR><FIRSTNAME>Charles</FIRSTNAME>
            <LASTNAME>Porter</LASTNAME></AUTHOR>
  </BOOK>
  <QUANTITY>6</QUANTITY>
  <DISCOUNT>.40</DISCOUNT>
 </ITEM>
 <ITEM>
  <BOOK><ISBN>15532155</ISBN>
    <TITLE>Introduction to Internet</TITLE>
    <PRICE>22.50</PRICE>
    <AUTHOR><FIRSTNAME>Steve</FIRSTNAME>
            <LASTNAME>Andrews</LASTNAME></AUTHOR>
  </BOOK>
  <QUANTITY>10</QUANTITY>
  <DISCOUNT>.42</DISCOUNT>
 </ITEM>
</ORDER>

<ORDER number=2>
 <SHIPTO>
  <FULLADDRESS><COMPANY>ASA</COMPANY><CITY>Los Angeles</CITY>
    <ADDRESSLINE>18 Harvard str.</ADDRESSLINE>
  </FULLADDRESS>
 </SHIPTO>
 <CONTACT><REFERENCE customer="C00002"></REFERENCE></CONTACT>
 <DATE><DAY>20</DAY><MONTH>11</MONTH><YEAR>1998</YEAR></DATE>
 <ITEM>
  <BOOK><ISBN>15536455</ISBN>
    <TITLE>Introduction to XML</TITLE>
    <PRICE>24.95</PRICE>
    <AUTHOR><FIRSTNAME>Charles</FIRSTNAME>
            <LASTNAME>Porter</LASTNAME></AUTHOR>
  </BOOK>
  <QUANTITY>6</QUANTITY>
  <DISCOUNT>.40</DISCOUNT>
 </ITEM>
 <ITEM>
  <BOOK><ISBN>15532155</ISBN>
    <TITLE>Introduction to Internet</TITLE>
    <PRICE>22.50</PRICE>
    <AUTHOR><FIRSTNAME>Steve</FIRSTNAME>
            <LASTNAME>Andrews</LASTNAME></AUTHOR>
  </BOOK>
  <QUANTITY>10</QUANTITY>
  <DISCOUNT>.42</DISCOUNT>
 </ITEM>
</ORDER>

<PERSON id="C00001">
 <FIRSTNAME>Robert</FIRSTNAME>
 <LASTNAME>Moore</LASTNAME>
 <FULLADDRESS><COMPANY>ABC</COMPANY><CITY>Los Angeles</CITY>
   <ADDRESSLINE>10 Michigan str.</ADDRESSLINE>
 </FULLADDRESS>
</PERSON>
<PERSON id="C00002">
 <FIRSTNAME>Tom</FIRSTNAME>
 <LASTNAME>Smith</LASTNAME>
 <FULLADDRESS><COMPANY>ASA</COMPANY><CITY>Los Angeles</CITY>
   <ADDRESSLINE>18 Harvard str.</ADDRESSLINE>
 </FULLADDRESS>
</PERSON>
<PERSON id="C00003">
 <FIRSTNAME>Steve</FIRSTNAME>
 <LASTNAME>Andrews</LASTNAME>
 <FULLADDRESS><CITY>San Francisco</CITY>
   <ADDRESSLINE>15 Washington str.</ADDRESSLINE>
 </FULLADDRESS>
</PERSON>
```

Figure 2: Running Example of XML Document

2.2 XML Graphical Data Model

The XML-GDM data model consists of three concepts: objects, relationships, and properties.

- *Objects*, depicted as rectangles, indicate abstract items without a directly representable value.
- *Properties*, depicted as circles connected to the object they refer to, indicate representable values (e.g., a character data or parsed character data string); properties have a name and a type, represented as labels.
- *Relationships*, depicted as arcs between objects, indicate semantic associations (e.g., containment or reference). Relationships have an orientation, from a source object to a destination object.

2.3 Representing XML DTDs and Documents

For the sake of explanation, we classify XML content into four categories:

- *Printable content*: PCDATA content[1].
- *Non-terminal elements*: XML elements which include other sub-elements.
- *Terminal elements*: XML elements with printable content or EMPTY content.
- *Mixed elements*: XML elements with either printable content or element content, in mutual exclusion.

XML attributes are also classified as *object-identifiers*, if their type is ID, *object-valued*, if their type is IDREF or IDREFS, or *printable* attributes, otherwise.

The correspondence between an XML DTD and an XML document and a XML-GDM graph is established by the following rules[2]:

- Each *non-terminal element* E is mapped to an XML-GDM object with the same name as E.
- Between any two non-terminal elements E1 and E2 such that E1 is a *sub-element* of E2 we establish a relationship from the object that represents E2 to the object that represents E1.
- When a *terminal element* E1 is a sub-element of another element E2, it is mapped to a property of E2 named E1, with PCDATA type. In the representation of an element occurrence in a document, this property has the same value as the XML PCDATA content. If a terminal element E is *not* contained in any other element, it is represented as an XML-GDM object, named E, with one (predefined) property named *content*, of type PCDATA[3].
- *Element disjunction (|)*: if a non-terminal element E contains sub-elements $E_1, \ldots, E_n$ that are in exclusive "or", i.e., only one of them can be present in E, then an arc is drawn which crosses the relationships between E and $E_1, \ldots E_n$, labeled "xor".[4]

[1] For simplicity, in the rest of the paper we will not distinguish between CDATA and PCDATA content.

[2] The rules assume that all the *parameter entities* used in the DTD (e.g., to represent sub-elements or attributes shared by several elements) have been expanded.

[3] In the sequel we will omit from representation the type label in case of PCDATA and ID.

[4] This notation is similar to AND-OR graphs.

- *Mixed Elements*: a mixed element E1 containing either PCDATA or a subelement E2 is represented as an object E1 including the disjunction of E2 and the predefined property *content*, introduced above. This notation considers the predefined property *content* as equivalent to a predefined element (e.g., named PCDATA) with a single property named *content* of type PCDATA.

- Each *printable attribute* and *object-identifier* of an element E is mapped to a property of the object that represents E, with the same name and type of the XML attribute. In the representation of an element occurrence in a document the property has the same value as the XML attribute. For distinguishing XML attributes from nested terminal elements, for the former we color in black the small circle of the property.

- *Object-valued attributes*: each object-valued attribute of an element declaration E in a DTD is mapped to a relationship from the object that represents E to a predefined XML-GDM object, named ANY, which represents any XML element (terminal or non-terminal). The attribute name in the DTD is mapped to the label of the relationship[5]. In an actual XML document, each object-valued reference from an occurrence of an element E is mapped into a relationship from this occurrence to the single element occurrence having the ID specified in the IDREF attribute of the instance of E. If the object-valued attribute is of type IDREFS, a set of such relationships is introduced.

- *Content model cardinality constraints*: the *iteration* operators '+', '*' and the *optionality* operator '?' used in the content model of DTDs are expressed as cardinality constraints on either relationships or properties; cardinality constraints have the following forms: (0:N) for *, (0:1) for ?, and (1:N) for +. If no operator is present, the cardinality constraint (1:1) is assumed, which is taken as default (and omitted from the representation). With object-valued attributes, the type IDREF correspond to the cardinality (1:1) - or (0:1) if the attribute is IMPLIED - while the type IDREFS correspond to the default cardinality (1:N), - or (0:N) if the attribute is IMPLIED.

- *Element order*: the actual or required order of appearance of sub-elements in a super-element is represented by ordering the arcs that represent the containment relationships counterclockwise, starting from the arc corresponding to the first sub-element, which is marked by a small trait.

According to the above rules, the DTD of Figure 1 is represented in XML-GDM as shown in Figure 3. For example, the non-terminal element *order*, which is declared in the DTD as:

```
<!ELEMENT order (shipto, contact?, item+, date)>
<!ATTLIST order number PCDATA #REQUIRED>
```

is represented by the XML-GDM subgraph including the nodes labeled *order, number, shipto, contact, item*, and *date*, rooted in node *order*. Object *order* has one property, corresponding to the attribute *number* of the related XML element. Four relationships connect *order* to its children elements. Relationships with *contact* and *item* are marked (0:1) and (1:N) respectively, to represent the use of operators ? and + in the content model. The relationship between *order* and *shipto* is marked as the first in the order, and other relationships are drawn counterclockwise from it in the same order as in the element declaration.

The non terminal element *contact*, which is declared in the DTD as:

[5]For uniformity, XML-GDM containment relationships stemming from element nesting also have a label (namely, *contains*), which can be omitted without ambiguity.

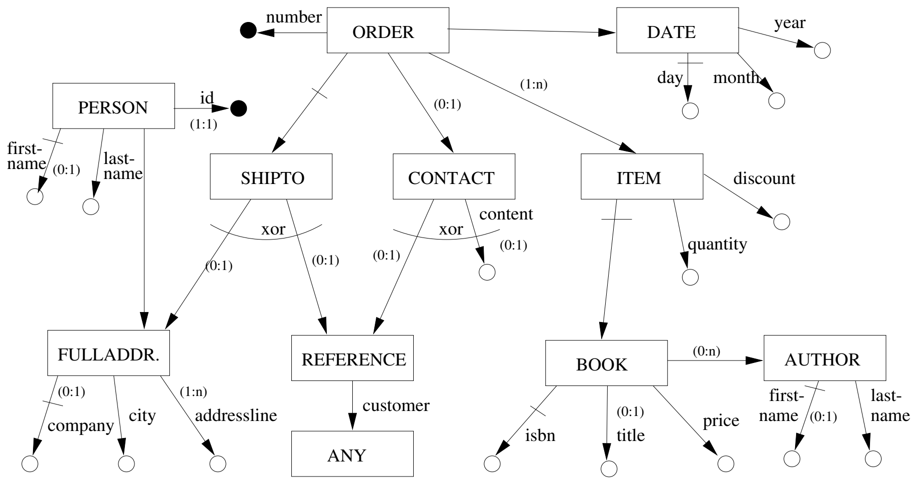

Figure 3: Example of DTD according to the XML-GML model.

```
<!ELEMENT contact (reference|PCDATA)>
```

has a mixed content model including either one element or PCDATA content; it is represented by an XML-GDM graph including one object *contact*, connected by a containment relationship to the object *reference* and with a property *content*; disjunction is represented as an "xor" edge crossing the containment relationship and the predefined property.

The non-terminal element *book*, which is declared in the DTD as:

```
<!ELEMENT book (isbn,title?,price,author*)>
<!ELEMENT author (firstname?,lastname)>
<!ELEMENT firstname PCDATA>
<!ELEMENT lastname PCDATA>
<!ELEMENT isbn PCDATA>
<!ELEMENT title PCDATA>
<!ELEMENT price PCDATA>
```

is represented by the XML-GDM subgraph including the nodes labeled *book, isbn, title, price and author*, rooted in node *book*. *book* and *author* are objects, connected by a relatioship, while the terminal elements *isbn, title, price, firstname, and lastname* are represented as properties.

XML-GDM supports the representation of actual XML documents with the same formalism as for DTDs, except that cardinality constraints and disjunction need not be represented, since a particular element occurrence always has a specific set of sub-elements and a particular choice of alternatives in the representations. A piece of the instance of Figure 2 is represented in Figure 4.

3 Query Language

XML-GL is a query language for XML-GDM data. An XML-GL query can be applied either to a single XML document or to a set of documents, e.g., those composing a WWW site. The query produces a new XML document as the result. Thus, the execution of a query results in a transformation of the source XML document(s) into a new XML document. An XML-GL query consists of four parts:

1. The *extract* part identifies the scope of the query, by indicating both the target documents and the target elements inside these documents; by drawing a parallel with SQL, the extract part

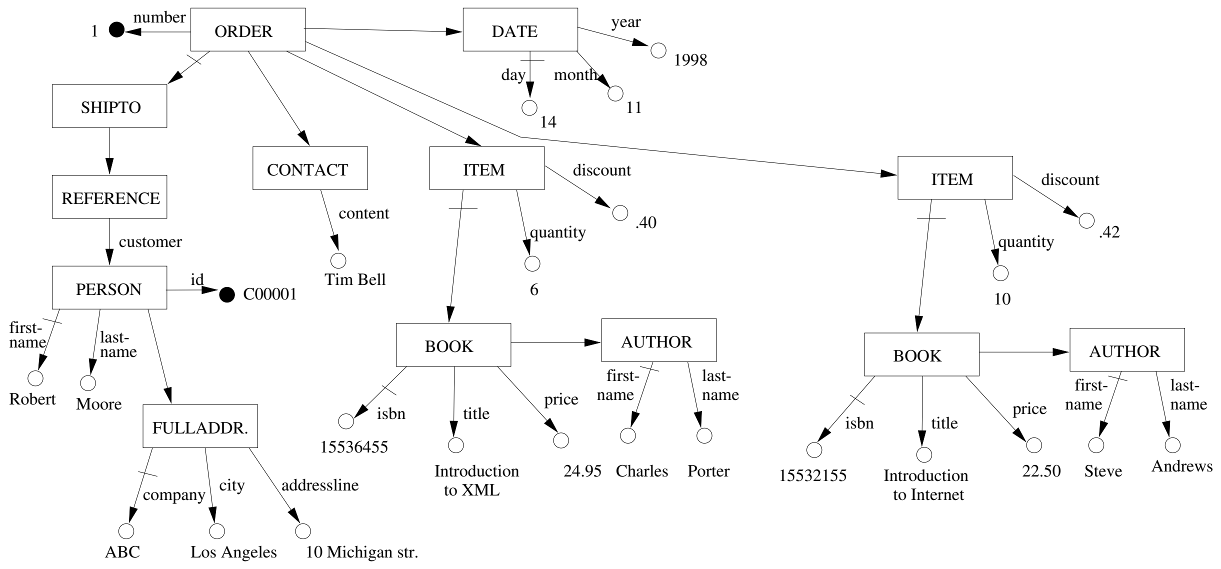

Figure 4: Representation of the document of the running example in XML-DML.

can be seen as the counterpart of the `from` clause, which establishes the relations targeted by the query.

2. The *match* part (optional) specifies logical conditions that the target elements must satisfy in order to be part of the query result; continuing the parallel with SQL, the condition part can be seen as the counterpart of the `where` clause, which chooses the target tuples that are part of the result.

3. The *clip* part specifies the sub-elements of the extracted elements that satisfy the match part to be retained in the result. With respect to SQL, the clip part corresponds to the `select` clause, which permits the user to define which columns of the result tuples should be retained in the final output of the query.

4. The *construct* part (optional) specifies the *new* elements to be included in the result document and their relationships to the extracted elements; the same query can be formulated with different contruction parts, to obtain results formatted differently. With respect to SQL, the construct part can be seen as the extension of the `create view` statement, which permits the user to design a new relation from the result of a query. The construct part permits both the creation of new elements, the definition of new links, and the restructuring of information local to a given element.

Graphically, a XML-GL query is a pair of XML-GDM graphs, displayed side by side and separated by a vertical line; the left-hand-side graph visually represents the extract and match parts, while the right-hand-side graph conveys the clip and construct parts. This separation sharply evidentiates those concepts which are used to extract elements from the target documents and those concepts which are used to construct the result documents produced by the query. Indeed, *the right graph represents the DTD of the result*; such DTD is always present, even if the query is applied to well-formed documents without a DTD.

In the following sections, we progressively introduce the features of XML-GL by means of sample queries with increasingly complex structures. First, we will show simple queries that extract elements from target documents and produce result documents in different ways (Extract-Clip queries, Section 3.1); then, we show queries that apply filtering conditions to the extracted elements to be included in the result (Extract-Match-Clip queries, Section 3.2); finally we will introduce queries that define arbitrarily structured result documents, composed of new elements, possibly intermixed with elements extracted from the target documents (Extract-Match-Construct-Clip queries, Section 3.3).

3.1 Extract-Clip Queries

The simplest form of XML-query is the Extract-Clip, which extracts a portion of an XML document and produces as output a new document containing the extracted data.

Example I: The query of Figure 5 *finds all the book elements from a specified set of documents over the WWW.* Its result is shown in Figure 7.b.

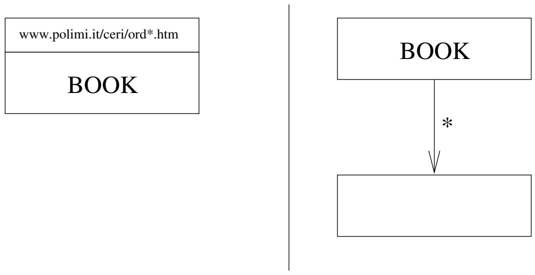

Figure 5: Example of extract-clip query.

In Extract-Clip queries, the left-hand-side graph contains the extract part of the query. In the example, the extract part operates on the single target element *book*. More generally, the extract part of a query may contain several target elements, represented as root nodes of the left-hand-side graph.

Target elements may optionally contain the indication of the URLs of the document or set of documents that should be used as input in order to evaluate the query. For convenience, URL in queries may contain wildcards; in the query of Figure 5, the string "http://www.elet.polimi.it/ceri/ord*.xml" inside object *book* makes the query target only those XML documents in the *ceri* directory of host "www.polimi.it", whose name starts with the string "ord".

The right-hand-side graph expresses the clip part, which defines the DTD of the result document as an XML-GDM graph, out of the structure of the target elements mentioned in the extract part. When an object mentioned in the extract part should belong to the result of the query, it must be included also in the clip part. The correpondence between left-hand-side and right-hand-side objects is by name (as for object *book* in the example of Figure 5); if this introduces ambiguity (e.g., for queries using the same elements multiple times in the extract part), then the one-to-one correspondence may be made explicit by drawing an edge that connects the corresponding elements of the two graphs.

The meaning of the one-to-one correspondence is that the result of the query will be constructed - according to the structure dictated by the right-side graph - using exactly those object instances which are selected by the extract part and are mentioned in the clip part. In the example of Figure 5, all books found in the target XML documents are used to build the result.

When an extracted element is used to build the result, the clip part must also specify which sub-elements should be retained and which should be discarded. To make the clip part more concise, the following shorthand notations are defined, represented in Figure 6:

- All sub-elements at the first level of nesting are kept (Figure 6.a);

- All sub-elements at all levels of nesting are kept (Figure 6.b);
- Only the terminal sub-elements and elements of type PCDATA at the first level of nesting are kept (Figure 6.c);
- Only the terminal sub-elements and elements of type PCDATA at all levels of nesting are kept (Figure 6.d);
- All occurrences of a given element found at any level of nesting, without the intermediate enclosing elements are kept (Figures 6.e and e').

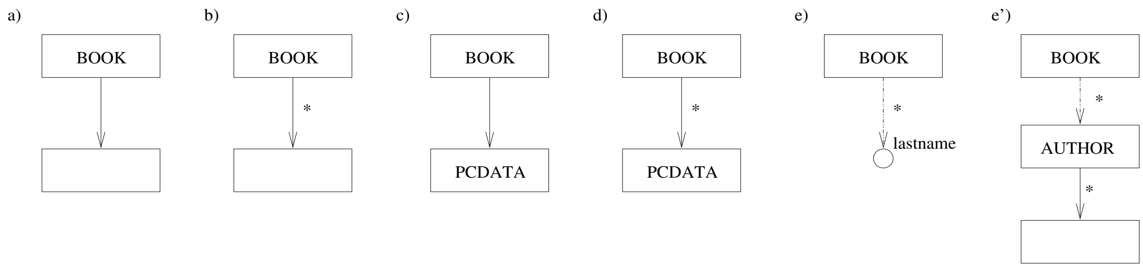

Figure 6: Graphic Notations for Expressing the Clip part.

The results produced by the above clip parts applied to the set of all books in the running example document are illustrated in Figure 7; in particular, the query in Figure 5 is represented by case (b).

3.2 Extract-Match-Clip Queries

The match part extends the left-hand-side graph of the query with the possibility of expressing a large class of selection predicates. These are described by means of a well-defined collection of graphic notations which enable the expression of existential conditions (e.g., the requirement that a subelement exists), or predicates on element's properties (e.g., required values for attributes and PCDATA content); all predicates are implicitly in conjunctive form.

The condition of a query normally involves several sub-elements of the target elements in the extract part; these elements are evidentiated in the left-hand-side graph. This operation is eased by the presence of the XML-GDM representation of the DTD of the input document(s), which exactly gives the needed graphical representation of the elements' internal structure. The same notation is also used to specify the subelements to be kept in the clip part, as shown in the previous section. Thus, query construction can be easily supported by a "drag-and-drop" interface, starting from the construction of the graph in the left part of a query, and then proceding to the right-hand-side part of the query.

Example II: The query of Figure 8 *finds orders containing the book titled "Introduction to XML", to be shipped to an address in Los Angeles, and presents such orders with their shipping and item information*[6].

The document produced as result of the previous query is the following:

[6]For the sake of simplicity, from now on we will omit URL in the match part .

Notation A: first level elements

```
<BOOK>
  <ISBN>15536455</ISBN>
  <TITLE>Introduction to XML</TITLE>
  <PRICE>24.95</PRICE>
  <AUTHOR></AUTHOR>
</BOOK>
<BOOK>
  <ISBN>15532155</ISBN>
  <TITLE>Introduction to Internet</TITLE>
  <PRICE>22.50</PRICE>
  <AUTHOR></AUTHOR>
</BOOK>
```

Notation B: all level elements

```
<BOOK>
  <ISBN>15536455</ISBN>
  <TITLE>Introduction to XML</TITLE>
  <PRICE>24.95</PRICE>
  <AUTHOR><FIRSTNAME>Charles</FIRSTNAME>
          <LASTNAME>Porter</LASTNAME></AUTHOR>
</BOOK>
<BOOK>
  <ISBN>15532155</ISBN>
  <TITLE>Introduction to Internet</TITLE>
  <PRICE>22.50</PRICE>
  <AUTHOR><FIRSTNAME>Steve</FIRSTNAME>
          <LASTNAME>Andrews</LASTNAME></AUTHOR>
</BOOK>
```

Notation C: first level terminal and PCDATA

```
<BOOK>
  <ISBN>15532155</ISBN>
  <TITLE>Introduction to Internet</TITLE>
  <PRICE>22.50</PRICE>
</BOOK>
<BOOK><ISBN>15536455</ISBN>
  <TITLE>Introduction to XML</TITLE>
  <PRICE>24.95</PRICE>
</BOOK>
```

Notation D: all level terminal and PCDATA

```
<BOOK>
  <ISBN>15536455</ISBN>
  <TITLE>Introduction to XML</TITLE>
  <PRICE>24.95</PRICE>
  <FIRSTNAME>Charles</FIRSTNAME>
  <LASTNAME>Porter</LASTNAME>
</BOOK>
<BOOK>
  <ISBN>15532155</ISBN>
  <TITLE>Introduction to Internet</TITLE>
  <PRICE>22.50</PRICE>
  <FIRSTNAME>Steve</FIRSTNAME>
  <LASTNAME>Andrews</LASTNAME>
</BOOK>
```

Notation E: all occurrences of a nested property

```
<BOOK>
  <LASTNAME>Porter</LASTNAME>
</BOOK>
<BOOK>
  <LASTNAME>Andrews</LASTNAME>
</BOOK>
```

Notation E': all occurrences of a nested element

```
<BOOK>
  <AUTHOR>
    <FIRSTNAME>Charles</FIRSTNAME>
    <LASTNAME>Porter</LASTNAME></AUTHOR>
</BOOK>
<BOOK>
  <AUTHOR>
    <FIRSTNAME>Steve</FIRSTNAME>
    <LASTNAME>Andrews</LASTNAME></AUTHOR>
</BOOK>
```

Figure 7: Results of Different Clip Parts for the Query of Figure 5

```
<ORDER number=2>
 <SHIPTO>
  <FULLADDRESS><COMPANY>ASA</COMPANY><CITY>Los Angeles</CITY>
    <ADDRESSLINE>18 Harvard str.</ADDRESSLINE>
  </FULLADDRESS>
 </SHIPTO>
 <ITEM>
  <BOOK><ISBN>15536455</ISBN>
    <TITLE>Introduction to XML</TITLE>
    <PRICE>24.95</PRICE>
    <AUTHOR><FIRSTNAME>Charles</FIRSTNAME><LASTNAME>Porter</LASTNAME></AUTHOR>
  </BOOK>
  <QUANTITY>6</QUANTITY>
  <DISCOUNT>.40</DISCOUNT>
 </ITEM>
 <ITEM>
  <BOOK><ISBN>15532155</ISBN>
    <TITLE>Introduction to Internet</TITLE>
    <PRICE>22.50</PRICE>
    <AUTHOR><FIRSTNAME>Steve</FIRSTNAME><LASTNAME>Andrews</LASTNAME></AUTHOR>
  </BOOK>
  <QUANTITY>10</QUANTITY>
  <DISCOUNT>.42</DISCOUNT>
 </ITEM>
</ORDER>
```

Note that all orders appearing in the result have both an address and (at least) one item satisfying the extract-match part.

The condition in the match part may also involve the application of boolean operators to attributes and PCDATA properties. To this end, the match part may use the comparison operators (>, <, >=, <=, = and <>) and the string operators (_ and %). The *match* part can also be used to write queries targeted to several elements, similar to select-join queries of SQL.

Example III: The query of Figure 9 *finds all books written by an author with the same lastname as a person whose name starts with 'S'.*

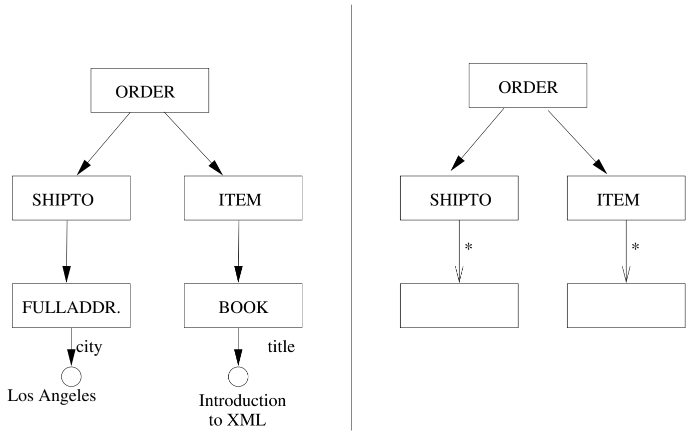

Figure 8: Example of Extract-Match-Clip query with a match part imposing predicates on sub-elements.

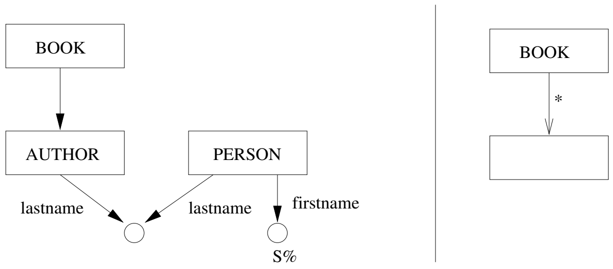

Figure 9: Example of Extract-Match-Clip query with join.

The join condition on lastnames is expressed in the match part by expanding the graph of the *book* element to show the inner *author* element, and connecting the author's and person's last names; note that for both the *author* and *person* elements *lastname* is not an XML attribute, but a piece of XML data.

Queries involving element identity and references between elements are easily represented, based on the treatment of element identity explained in Section 2. Element identity is defined when an element has an *ID* attribute and other elements refer to it using *IDREF(S)* attributes: in this case a query may search for IDs that "point" to the same element, i.e. that have the same value, as demonstrated in the following example.

Example IV: The query of Figure 10 *finds the persons referenced as contact in an order.*

The query extracts those orders containing a contact that includes a reference (order #2, in the running example), and pairs them to the person whose ID matches the *customer* attribute of the reference element. These persons are then included in the result in the clip part.

Element identity can also be used in join conditions: the following query exploits IDREF attributes to "join" information of orders.

Example V: The query of Figure 11 *finds the orders shipped to persons who are also contacts in another order* .

Note that in this case the extract-match part of the query contains *two* ORDER elements, and ambiguity is avoided by explicitly connecting only one of them to its counterpart in the clip part. This technique compares with the use of alias (with the keyword AS) in SQL queries in order to

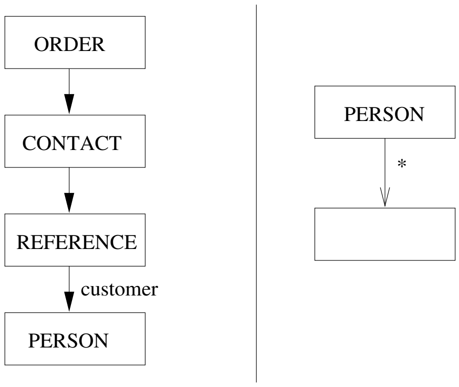

Figure 10: Example of Extract-Match-Clip query involving ID reference.

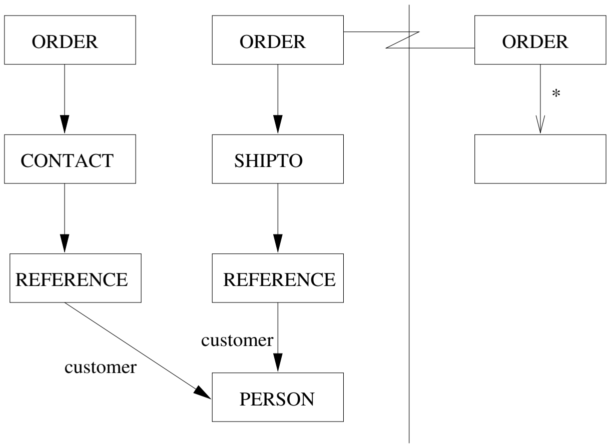

Figure 11: Example of Extract-Match-Clip query using element identity.

disambiguate multiple occurrences of the same table or column name by associating each of them to a different variable.

Both positive and negated conditions are admitted: to express that a condition is negative, this is drawn using dashed lines, as in the following example.

Example VI: The query of Figure 12.a *finds all books having a title*, while the query of Figure 12.b *finds the books with unknown title.*

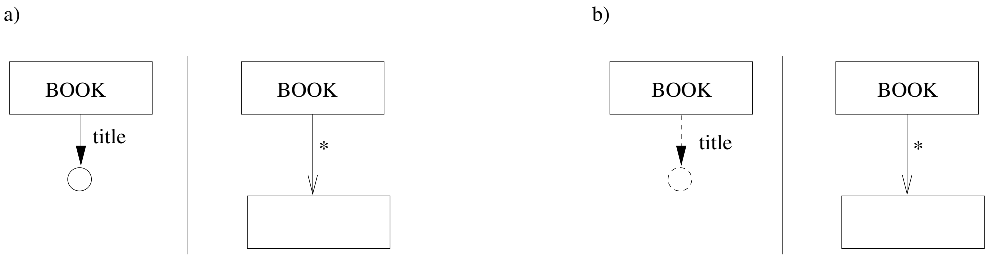

Figure 12: Examples of Extract-Match-Clip queries with positive (a) and negated (b) conditions in the match part.

In the preceding examples, conditions in the match part always specified the name of the element to be extracted/matched: however, if a query requires to express a condition on a generic object, *dummy nodes* are used. They are represented as unlabeled nodes which are matched against ANY element.

Example VII: The query of Figure 13 *finds all the elements that contain a book and includes them in the result with their PCDATA content and terminal sub-elements.*

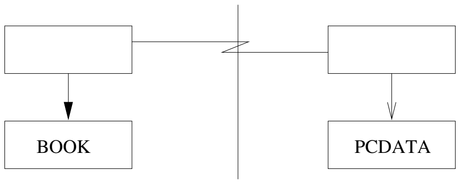

Figure 13: Example of Extract-Match-Clip query with the special object ANY.

This query produce as result the following XML data:

```
<ITEM><QUANTITY>6</QUANTITY><DISCOUNT>.40</DISCOUNT></ITEM>
<ITEM><QUANTITY>10</QUANTITY><DISCOUNT>.42</DISCOUNT></ITEM>
<ITEM><QUANTITY>6</QUANTITY><DISCOUNT>.40</DISCOUNT></ITEM>
<ITEM><QUANTITY>10</QUANTITY><DISCOUNT>.42</DISCOUNT></ITEM>
```

3.3 Extract-Match-Construct-Clip Queries

So far, XML-GL queries have produced very simple result documents, defined by a subset of the elements extracted from target documents in the extract part. However, XML-GL can be used to produce more sophisticated result documents which:

- combine sub-elements of several target elements;
- introduce new elements whose content can be derived from that of existing elements extracted from the target documents;
- contain new elements providing content grouping or reordering capabilities.

From a document processing point of view, the semantics of the construct part of XML-GL queries is similar to that of a *transformation program* that converts a tagged document into another one by means of pattern matching and rewriting, as proposed for instance in the *DSSSL (Document Style Semantics and Specification)* language [Pre98]. However, while DSSSL provides its transformation language within a stylesheet-based environment for rendering and processing SGML documents, XML-GL Construct feature is concerned with restructuring alone, cleanly separating the transformation from the presentation issues. This corresponds to the functional decomposition envisioned in recent proposals for XML-based WWW application environments [MUT98]. Indeed, XML-GL Construct can be considered as a complement to current capabilities of XSSL (XML Style Sheet Language) [W3C98], which again is mainly concerned with XML documents rendering.

3.3.1 Embedding Extracted Content into New Elements

The simplest form of construction consist of embedding elements extracted in the extract-match part into new elements. Three types of embedding are possible:

- *Constructed element*: each element extracted by the extract-match part is embedded into a distinct instance of a new element. Element construction is denoted by a containment relationship between the new element (represented as an XML-GDM object) and its sub-elements in the clip-construct part (See Figure 14.a).

- *List*: all elements extracted by the extract-match part are embedded inside *one* new element. List construction is denoted by a triangle representing the new element connected by a containment relationship to the objects in the clip-construct part representing the sub-elements to be nested (See Figure 14.b).

- *Grouping list*: occurrences of the same element extracted by the extract-match part are embedded inside multiple lists defined by a grouping criterion. Grouping list construction is denoted by an index (a rectangle with horizontal lines) representing the new grouping list connected by a containment relationship to the objects in the clip-construct part representing the sub-elements to be nested (see Figure 14.c). The grouping criterion is represented by an edge connecting the index to one or more elements used for grouping.

Example VIII: Consider the three queries of Figure 14. They find all the existing persons that have an address; with element construction (a), one instance of the new element (called RESULT) is created for each person satisfying the given condition and contains the person's data according to the clip specification; with the list construction (b) a single element (also called RESULT) is created which contains the list of persons satisfying the match part, along with their nested subelements as specified by the clip specification; with the grouping list construction (c) the resulting persons are grouped by city and one element (named RESULT) is introduced for each group. Formally, *city* induces a partition of the occurrences of *person*, where each distinct value of *city* is mapped to a distinct subset of persons.

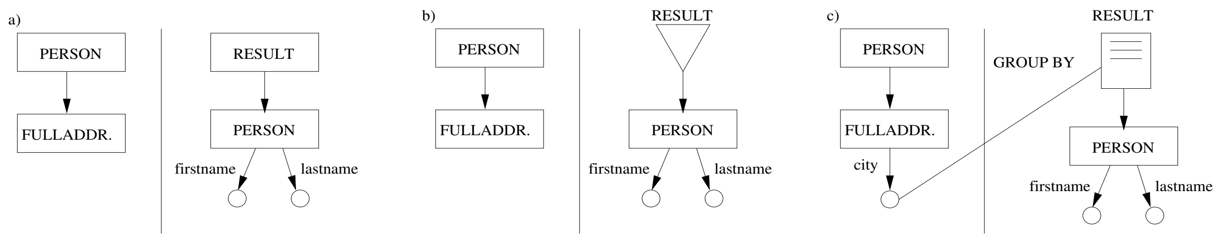

Figure 14: Examples of constructed element(a), list (b), and grouping list (c).

Thus, the results of the three queries applied to the document of Figure 2 are the following:

Query A: element construction

```
<RESULT>
  <PERSON id="C00001">
  <FIRSTNAME>Robert</FIRSTNAME>
   <LASTNAME>Moore</LASTNAME>
  </PERSON>
</RESULT>
<RESULT>
  <PERSON id="C00002">
   <FIRSTNAME>Tom</FIRSTNAME>
   <LASTNAME>Smith</LASTNAME>
  </PERSON>
</RESULT>
<RESULT>
  <PERSON id="C00003">
   <FIRSTNAME>Steve</FIRSTNAME>
   <LASTNAME>Andrews</LASTNAME>
  </PERSON>
</RESULT>
```

Query B: list construction

```
<RESULT>
  <PERSON id="C00001">
   <FIRSTNAME>Robert</FIRSTNAME>
   <LASTNAME>Moore</LASTNAME>
  </PERSON>
  <PERSON id="C00002">
   <FIRSTNAME>Tom</FIRSTNAME>
   <LASTNAME>Smith</LASTNAME>
  </PERSON>
  <PERSON id="C00003">
   <FIRSTNAME>Steve</FIRSTNAME>
   <LASTNAME>Andrews</LASTNAME>
  </PERSON>
</RESULT>
```

Query C: grouping list construction

```
<RESULT>
  <PERSON id="C00001">
  <FIRSTNAME>Robert</FIRSTNAME>
  <LASTNAME>Moore</LASTNAME>
  </PERSON>
  <PERSON id="C00002">
   <FIRSTNAME>Tom</FIRSTNAME>
   <LASTNAME>Smith</LASTNAME>
  </PERSON>
</RESULT>
<RESULT>
  <PERSON id="C00003">
   <FIRSTNAME>Steve</FIRSTNAME>
   <LASTNAME>Andrews</LASTNAME>
  </PERSON>
</RESULT>
```

Figure 15 pictorially summarizes the three different construction primitives and how they build the result from the extracted elements. The first option lists all the selected elements, the second option builds a result element containing all the selected persons, and the third option presents them according to the partitioning produced by the grouping criterion.

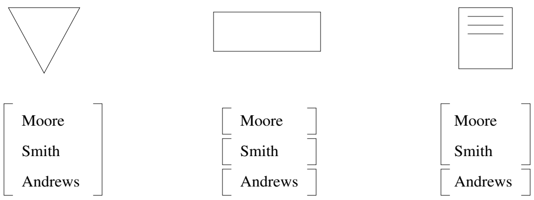

Figure 15: Summary of construction primitives.

Example IX: The query of Figure 16 demonstrates the orthogonal combination of construction primitives; it is a variant of Figure 14.c: it *groups persons by city and embeds these groups into a new element called RESULT containing also the name of the city.*

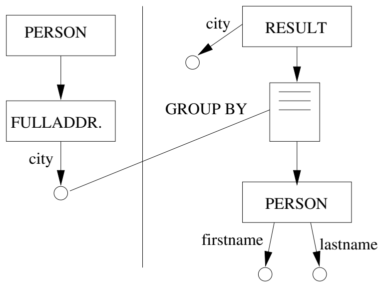

Figure 16: Example of orthogonal combination of construction primitives.

The query applied to the document of Figure 2 returns the following result:

```
<RESULT><CITY>Los Angeles</CITY>
  <PERSON id="C00001">
   <FIRSTNAME>Robert</FIRSTNAME><LASTNAME>Moore</LASTNAME>
  </PERSON>
  <PERSON id="C00002">
   <FIRSTNAME>Tom</FIRSTNAME><LASTNAME>Smith</LASTNAME>
  </PERSON>
</RESULT>
<RESULT><CITY>San Francisco</CITY>
  <PERSON id="C00003">
   <FIRSTNAME>Steve</FIRSTNAME><LASTNAME>Andrews</LASTNAME>
  </PERSON>
</RESULT>
```

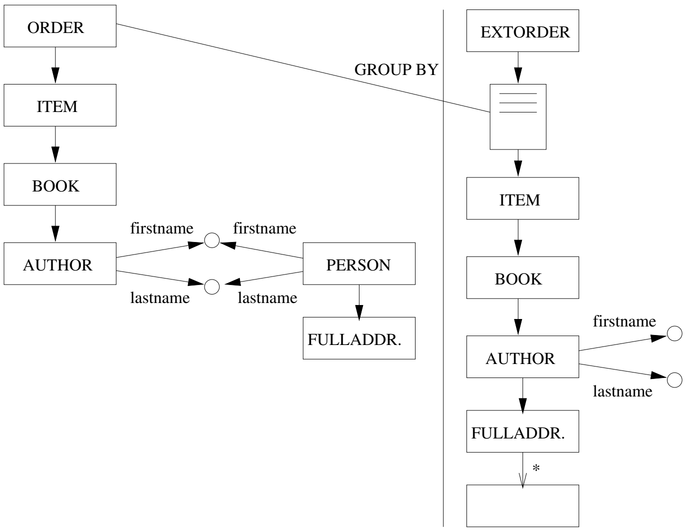

Figure 17: Example of extension of an element in the construct part.

3.3.2 Extension of an Element

XML-GL can also be used to restructure existing documents, e.g., by including elements of one document into another one or extending the elements of one document with information coming from related elements inside the same document.

Example X: The query of Figure 17 *finds the orders containing a book whose author's firstname and lastname appear also in an element of type* person *and produces a new element* extorder *where the address is added to each author.*

The result of this query applied on the document of Figure 2 is the following:

```
<EXTORDER>
 <ITEM>
  <BOOK><AUTHOR>
         <FIRSTNAME>Steve</FIRSTNAME><LASTNAME>Andrews</LASTNAME>
         <FULLADDRESS><CITY>San Francisco</CITY>
            <ADDRESSLINE>15 Washington str.</ADDRESSLINE>
         </FULLADDRESS>
        </AUTHOR>
  </BOOK>
 </ITEM>
</EXTORDER>
```

```
<EXTORDER>
 <ITEM>
  <BOOK><AUTHOR>
         <FIRSTNAME>Steve</FIRSTNAME><LASTNAME>Andrews</LASTNAME>
         <FULLADDRESS> <CITY>San Francisco</CITY>
            <ADDRESSLINE>15 Washington str.</ADDRESSLINE>
         </FULLADDRESS>
        </AUTHOR>
  </BOOK>
 </ITEM>
</EXTORDER>
```

In the result, one element EXTORDER is constructed for each group of items belonging to the same order retrieved in the extract-match part. Items in the result are the same as those retrieved in the extract-match (as the by-name correspondence indicates), but for a fact: each AUTHOR sub-element is extended with the inclusion of the FULLADDRESS element coming from the corresponding PERSON object retrieved in the extract-match part.

3.4 Unnesting and Nesting of an Element

XML-GL can be used to reproduce common operations on complex objects typical of object-oriented and nested relational databases. The following two queries demonstrate how *nesting* and *unnesting* is achieved. With nesting, several flat objects with a common property P can be restructured into a single object containing multiple values of P. Unnesting is the reverse operation that takes a single complex object containing a multi-valued property P and produces one simple object for each value of P.

Example XI: The query of Figure 18 *finds the orders having a* shipto *and an* item *element and a* number *attribute, and produces a flat list of triples each containing an order with its order number, shipping information and the book title of one of its items.*

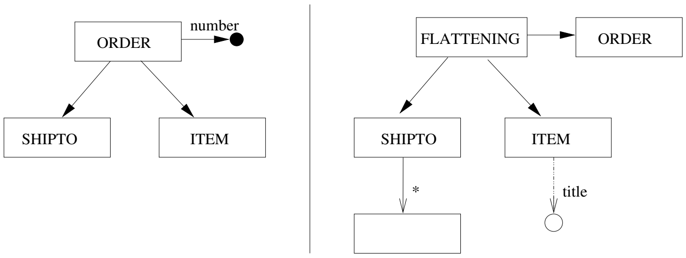

Figure 18: Example of unnesting of an element in the construct part.

The query applied to the document of Figure 2 produces the following result:

```
<FLATTENING>
 <ORDER number=1></ORDER>
 <SHIPTO><REFERENCE customer="C00001"></REFERENCE></SHIPTO>
 <ITEM><TITLE>Introduction to XML</TITLE></ITEM>
</FLATTENING>
<FLATTENING>
 <ORDER number=1></ORDER>
 <SHIPTO><REFERENCE customer="C00001"></REFERENCE></SHIPTO>
 <ITEM><TITLE>Introduction to Internet</TITLE></ITEM>
</FLATTENING>
```

```
<FLATTENING>
 <ORDER number=2></ORDER>
 <SHIPTO>
  <FULLADDRESS><COMPANY>ASA</COMPANY><CITY>Los Angeles</CITY>
    <ADDRESSLINE>18 Harvard str.</ADDRESSLINE>
  </FULLADDRESS>
 </SHIPTO>
 <ITEM><TITLE>Introduction to XML</TITLE></ITEM>
</FLATTENING>
<FLATTENING>
 <ORDER number=2></ORDER>
 <SHIPTO>
  <FULLADDRESS><COMPANY>ASA</COMPANY><CITY>Los Angeles</CITY>
    <ADDRESSLINE>18 Harvard str.</ADDRESSLINE>
  </FULLADDRESS>
 </SHIPTO>
 <ITEM><TITLE>Introduction to Internet</TITLE></ITEM>
</FLATTENING>
```

Example XII: The query of Figure 19 does the inverse operation: it extracts information from a flat set of elements and builds new structured orders, each of them containing the group of all items associated to the same order and the corresponding shipment information.

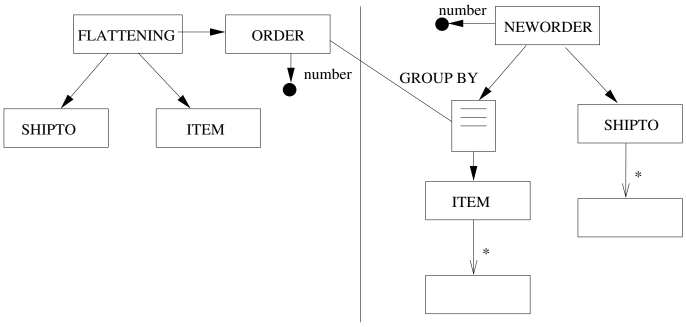

Figure 19: Example of nesting of an element in the construct part.

The query of Figure 19 applied to the unnested result of the previous query produces the following result:

```
<NEWORDER number=1>
 <SHIPTO><REFERENCE customer="C00001"></REFERENCE></SHIPTO>
 <ITEM><TITLE>Introduction to XML</TITLE></ITEM>
 <ITEM><TITLE>Introduction to Internet</TITLE></ITEM>
```

```
</NEWORDER>
<NEWORDER number=2>
 <SHIPTO>
  <FULLADDRESS><COMPANY>ASA</COMPANY>
   <ADDRESSLINE>18 Harvard str.</ADDRESSLINE>
   <CITY>Los Angeles</CITY>
  </FULLADDRESS>
 </SHIPTO>
 <ITEM><TITLE>Introduction to XML</TITLE></ITEM>
 <ITEM><TITLE>Introduction to Internet</TITLE></ITEM>
</NEWORDER>
```

4 XML-GL Advanced Features

Several advanced features have been proposed in the context of other XML query languages, like XML-QL [DFF+98], such as: order-sensitive queries, result sorting, arithmetics and aggregate functions. Quite nicely, these can be introduced into XML-GL orthogonally to the basic feature of the language, greatly enhancing its expressive power.

4.1 Order in the Match Part

Queries shown so far did not take into account the ordering of elements within XML documents. However, such ordering may be specified in the DTD and is uniquely defined in actual documents. XML-GL queries may take into account ordering, by extracting a given occurrence only if it presents its elements in the exact order dictated by the query. Graphically, we express that a query (or that a single portion of a query, circumscribed to one or more elements) is order-sensitive by using the same notation as in XML-GDM, as illustrated by the following example.

Example XIII: The query of Figure 20 *finds the persons whose lastname precedes their firstname.*

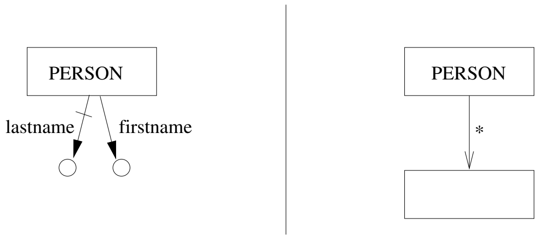

Figure 20: Example of query with explicit element ordering.

4.2 Order and Sorting in the Construct Part

While ordering in the left graph gives a more restrictive interpretation of the corresponding query, ordering in the right-hand-side graph specifies the order in which elements must be rearranged in the result. This is achieved by the same notation as in the match part. Moreover, in the construct part it is possible to specify if the occurrences of an element have to be sorted in ascending or descending order based on the values of some properties. The involved properties are labeled with the desired sort criterion (ASC, DESC), and, if element sorting requires multiple properties to be considered (e.g. lastname followed by firstname), the involved properties must be ordered and labeled with the sort criterion.

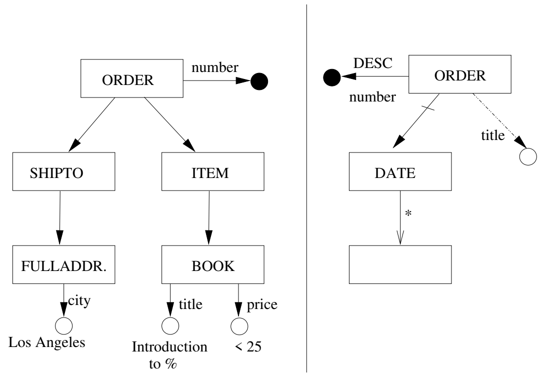

Figure 21: Example of query with order and sorting in the construct part.

Example XIV: The query of Figure 21 *returns as output the orders satsfying the extract-match part, sorted in descending order number and internally organized so that the date is placed before the book title.*

4.3 Queries with Arithmetic Functions in the Match Part

Conditions can be also imposed on values obtained by applying arithmetic operators on numeric values retrieved in the match part. The arguments are connected by an arc labeled with the type of operation and this arc is connected to a new property that contains the result: conditions can be imposed on the value of this result. The usual operators are provided (e.g. '+', '*', '-', '/').

Example XV: The query of Figure 22 *finds all items for which the product of the price and the quantity is less than 100.*

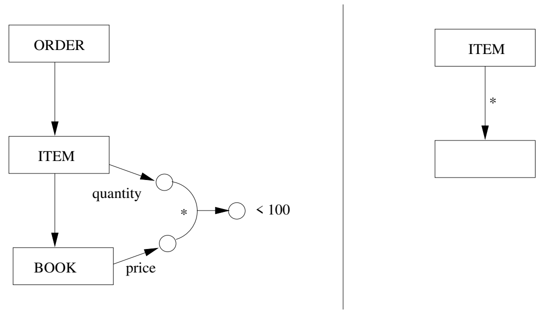

Figure 22: Example of Extract-Match-Clipt with a condition on a calculated field.

4.4 Queries with Aggregate Functions in the Match Part

It is also possible to apply aggregate functions such as sum, count, min, max and so on to groups of homogeneous elements to write conditions on aggregate values in the match part. Aggregation is expressed by means of the same notation used for specifying grouping in the construct part. An aggregate value is represented as a property of a grouping list; such property has the same name as the aggregate operator used in the calculation.

Example XVI: The query of Figure 23 *selects all the orders that have at least two items.*

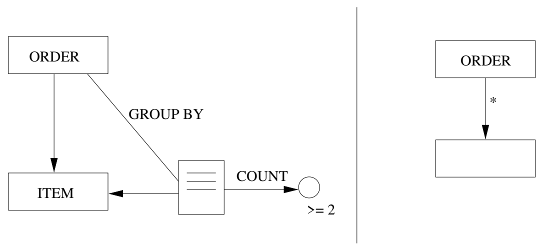

Figure 23: Example of match part with grouping and a condition on the aggregate value.

4.5 Queries with Arithmetic and Aggregate Functions in the Construction Part

As in SQL, in XML-GL arithmetics can be used not only to write conditions, but also to produce computed values in the result, as shown by the follwing example.

Example XVII: The query of Figure 24 *extends items with a new element* totbookprice*, calculated as the product of the price and the quantity of the individual item.*

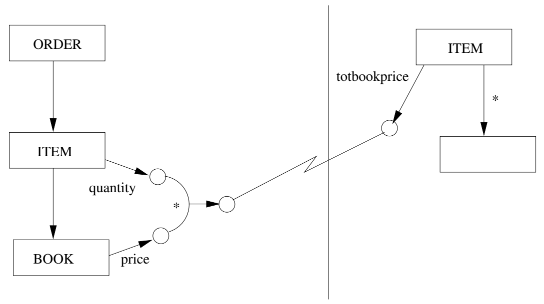

Figure 24: Example of Extract-Match-Clip with a calculated field in the construct part.

Grouping, aggregate functions and flexible document construction can be used orthogonally, to achieve complex document restructuring through simple and self-evident visual queries.

4.6 Notes on the Semantics of Construction

Though extensive discussion on the semantics of XML-GL is beyond the scope of this paper, we conclude the explanation of the language features with a remark on the semantics of the construction part. The correspondence between the graphical representation of the construct part of XML-GL queries and the document transformations needed to produce the result can be briefly illustrated with the help of the example of Figure 24.

As far as document transformation is concerned, executing the construct part of the query in Figure 24 amounts to:

- for each `<ITEM>` element in the document defined by the extract-match part of the query
- add inside `<ITEM>` (at the first level of nesting) a new element tag `<TOTBOOKPRICE>`,
- insert `<TOTBOOKPRICE>` content by computing (after suitable type conversion) the product of the content of element `<QUANTITY>` (located at the first nesting level inside the same `<ITEM>`) times the content of `<PRICE>` (which, in turn, is located at the second level of nesting, inside `<BOOK>`).

Using a syntax loosely inspired to DSSSL, the transformation program associated to the construct part of the XML-GL query could be expressed as follows:

```
<ITEM>
\{
make newelement AsChild
<TOTBOOKPRICE>
'<QUANTITY>.AsInteger
'<BOOK><PRICE>.AsInteger
*
</TOTBOOKPRICE >
\}
```

This sample transformation program opens with a *pattern* (`<ITEM>` in this case), to be matched inside the target document. The `make` action (whose argument is the new tag `<TOTBOOKPRICE>` specifies the rewriting to be performed whenever a match is found. A locality clause (*AsChild*) specifies where the new element should be put with respect to the searched pattern (immediately nested, in this case). Finally, the content of the new tag is specified by a Reverse Polish Notation expression taking the content of two previous patterns as operands; note that this expression can be straightforwardly deduced from the dataflow notation adopted in the construct part of the XML-GL query.

5 Related Work

The huge amount of data published via the World Wide Web has led to a number of research efforts on techniques to index, query and restructure WWW sites' contents. In this section we provide a brief overview of related work on XML query languages and, more generally, on query languages for the Web (see also [FLM98]).

A considerable amount of research has been made on how to complement keyword-based *searching* with database-style support for *querying* the Web. Several projects addressed this problem, and three main WWW query languages have been proposed so far: Web3QL [KS95], WebSQL [MMM96] and WebLog [LSS96]. The first two languages are modelled after standard SQL used for RDBMS, while the third retains the flavour of the Datalog language.

In the specific domain of XML documents, proposals for query languages are in their infancy. Several preliminary contributions and position papers are collected in [Wor98]. Among the discussed approaches we review XML-QL [DFF+98], XQL by Microsoft, Texcel, and webMethods, [RLS98], XQuery by Inso [DeR98], and XQL [IKK98] by Fujitsu.

The XML-QL language [DFF+98] has been submitted for evaluation to the WWW Consortium by a pool of researchers. XML-QL provides a textual syntax for writing queries that construct new XML documents from target documents. The expressive power of XML-QL is comparable to that of XML-GL, but the former has a different syntactic flavor based on the use of pattern-matching expressions and variables ranging over content and tag names to extract content from target documents and embed it into the result of a query.

The XML Query Language (XQL) [RLS98] is a notation proposed by several companies for addressing and filtering the elements and text of XML documents. XQL is an extension to the XSL pattern syntax [W3C98]. The basic idea is to provide a syntax to locate nodes (elements and text) within an XML document, using a notation inspired by directory path expressions. XQL relies on path expressions, filters, and methods to achieve an effect similar to the extract-match part of a XML-GL query. Conversely, there is no counterpart in XQL for the construction of new documents, as provided by the clip-construct part of XML-GL queries.

XQuery [DeR98] is a query language proposed by Inso Corporation for extracting information from XML documents. XQuery draws its syntax from the XPointer document linking language [MD98]. XQuery provides a rich type system for representing XML content and defines the output of queries as *locations*, which can be sets of XML elements, attributes or spans of text. A query consists of a sequence of steps: each step selects nodes, either in absolute terms or based on the output of the preceding step. Example of steps are: "selects the element with the given id" or "selects from among the direct children of the current location". Sequence of steps are joined bu the dot operator, like in OQL path expressions. XQuery most advanced features address the use of links in queries and of regular expressions to write order-sensitive queries.

XQL [IKK98] by Fujitsu takes a different approach to XML document querying, by proposing a syntax which extends well-known database query languages (SQL and OQL) to address the features of XML data. XQL has a select-from-where construct extended with tag variables, path expressions, and URL specification. XQL also includes primitive for the construction of output documents (inclusive of grouping) comparable to the construct-clip part of XML-GL queries.

6 Conclusions

XML-GL is a sophisticated, but intuitive, visual language for querying XML data sources. It draws its unique features from an original combination of orthogonal, natural primitives for visualizing DTDs and documents, extracting their content, producing new content from extracted data, and formatting query results in complex ways. The use of a visual interface and language for querying XML-based WEB documents seems very appealing.

Our future research activity will concentrate on the following directions. We will first consolidate the language and design a textual version with equivalent expressive power; we are interested in using a consensual query language jointly designed by the research community, if this will support our graphical constructs in a natural way. We will also address the requirements not currently satisfied by our proposal, such as: queries over arbitrarily linked documents and metadata, and flexible query interpretation and expansion for non-exact document matching. At the same time, we will give a precise definition of the semantics of the language; to this purpose, we will capitalize on our previous experience in designing G-Log [PPT95], a visual, logic-based language for querying semi-structured objects. Finally, we will concentrate on the deployment of the language within a WEB-based visual environment, by studying an effective query interface supporting the automatic display of DTDs and/or documents and a collection of graphic primitives for clipping and dragging schema elements and for incrementally constructing the query graphs.

References

[BGL98] Dan Brickley, R.V. Guha, and Andrew Layman. W3c rdf schemas (working draft), Oct. 1998. http://www.w3.org/TR/WD-rdf-schema/.

[CDQT98] A. Cortesi, A. Dovier, E. Quintarelli, and L. Tanca. Operational and abstract semantics of a query language for semi-structured information. In *Proc.of the Intl. Workshop on Deductive Logic Programming (DDLP '98)*, 1998.

[Con98] W3C Consortium. Xml 1.0, Feb. 1998. http://www.w3.org/XML.

[DeR98] Steven J. DeRose. Xquery: A unified syntax for linking and querying general xml documents. In *Query Languages 98* [Wor98].

[DFF+98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. Xml-ql: A query language for xml. In *Proc. QL'98 - The Query Languages Workshop*, Cambridge, Mass., Dec. 1998. http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/.

[FLM98] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the www: A survey. *SIGMOD Record*, 1998.

[IKK98] Hiroshi Ishikawa, Kazumi Kubota, and Yasuhiko Kanemasa. Xql: A query language for xml data. In *Query Languages 98* [Wor98].

[KS95] D. Konopnicki and O. Shmueli. W3ql: A query system for the world wideweb. In *Proc. of the 21th Intl. Conf. on Very Large Databases*, Zurich, 1995.

[LSS96] L. Lakshmanan, F. Sadri, and I. Subramanian. A declarative language for querying and restructuring the web. In *Proc. RIDE-NDS*. IEEE Computer Soc. Press, 1996.

[MD98] Eve Maler and Steve DeRose. Xml pointer language (xpointer), Mar. 1998. http://www.w3.org/TR/WD-xptr.

[MMM96] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. In *Proc. of the Conf. on Parallel and Distributed Information Systems*, Toronto, Canada, 1996.

[MUT98] H. Maruyama, N. Uramoto, and K. Tamura. Xml purpose and use in web applications, 1998. http://www.software.ibm.com/xml.

[PPT95] J. Paredaens, P. Peelman, and L. Tanca. G-log a declarative graph-based language. *IEEE Trans. on Knowledge and Data Eng.*, (7):436:453, 1995.

[Pre98] P. Prescod. An introduction to dsssl, 1998. http://itrc.uwaterloo.ca/papresco/dsssl/tutorial.html.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. Xml query language (xql). In *Query Languages 98* [Wor98].

[SERL98] Comai S., Damiani E., Posenato R., and Tanca L. A schema-based approach to modeling and querying www data. In *Proc. of FQAS'98*, Roskilde, May 1998. LNAI 1495.

[W3C98] W3C. An introduction to xsl, 1998. http://www.w3C.org/Style/xssl.

[Wor98] World Wide Web Consortium. *Query Languages 98*, Cambridge, Mass., Dec. 1998.