XQBE (*XQ*uery *B*y *E*xample): A Visual Interface to the Standard XML Query Language

DANIELE BRAGA, ALESSANDRO CAMPI, and STEFANO CERI
Politecnico di Milano

The spreading of XML data in many contexts of modern computing infrastructures and systems causes a pressing need for adequate XML querying capabilities; to address this need, the W3C is proposing XQuery as the standard query language for XML, with a language paradigm and a syntactic flavor comparable to the SQL relational language. XQuery is designed for meeting the requirements of skilled database programmers; its inherent complexity makes the new language unsuited to unskilled users.

In this article we present XQBE (XQuery By Example), a visual query language for expressing a large subset of XQuery in a visual form. In designing XQBE, we targeted both unskilled users and expert users wishing to speed up the construction of their queries; we have been inspired by QBE, a relational language initially proposed as an alternative to SQL, which is supported by Microsoft Access. QBE is extremely successful among users who are not computer professionals and do not understand the subtleties of query languages, as well as among professionals who can draft their queries very quickly.

According to the hierarchical nature of XML, XQBE's main graphical elements are trees. One or more trees denote the documents assumed as query input, and one tree denotes the document produced by the query. Similar to QBE, trees are annotated so as to express selection predicates, joins, and the passing of information from the input trees to the output tree.

This article formally defines the syntax and semantics of XQBE, provides a large set of examples, and presents a prototype implementation.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; H.2.3 [**Database Management**]: Languages—*Data manipulation languages (DML)*; D.1.7 [**Programming Techniques**]: Visual Programming

General Terms: Design, Languages

Additional Key Words and Phrases: Human interfaces, semi-structured data, XQuery, visual query languages, XML, visual query languages

Stefano Ceri is supported by CNR and by the MAIS, WEBSI, and CINQ grants; Daniele Braga and Alessandro Campi are supported by the Virtual Campus project sponsored by Microsoft.
Authors' address: Dipartimento di Elletronica e Informazione, Politecnico di Milano, Via Ponzio, 34/5, 20133 Milano, Italy; email: {braga,campi,ceri}@elet.polimi.it.

1. INTRODUCTION

The W3C (World Wide Web Consortium) provides two standard textual languages to express XML document transformations and to query XML data, XSLT [W3C 2001] and XQuery [W3C 2003b].[1] XQuery is gaining increasing popularity among computer scientists, especially those with an SQL background; indeed, the formulation of queries in XQuery and an SQL requires comparable "programming" skills. However, this nucleus of programmers is not vast if compared with the wide spectrum of applications in which XML is currently used. Many XML users need to query XML data—maybe for simple purposes and only occasionally—but they do not master full fledged and rather complicated languages such as XQuery or XSLT.

This article describes XQBE (*XQ*uery *B*y *E*xample), a user friendly XML query language based on a *visual* paradigm. The QBE paradigm (*Q*uery *B*y *E*xample [Zloof 1977]) demonstrated that a visual language is effective in supporting intuitive query expressions; effectiveness is high when the visual formalism matches the underlying data model and language in terms of basic constructs, involved querying paradigm, and visual abstraction. Accordingly, while QBE is a relational query language, based on the use of tables, XQBE is based on the use of trees, so as to adhere to the hierarchical XML data model.

1.1 Design Principles

XQBE was designed with the main objective of being easy to use; we also tried to make it highly expressive and directly mappable to XQuery, so that it can support GUIs capable of running on top of any existing XQuery implementation. Of course such goals cannot be fully achieved at the same time; *usability* is the most critical success factor, and therefore has been taken into consideration during the whole language design and GUI implementation process. Nevertheless, a visual representation for a complex transformation is inherently prone to becoming unreadable as the number of nodes grows higher; XQBE does best with simple transformations and we discourage its use for extremely complex transformations.

1.1.1 *Visual Query Paradigm.* Figure 1 summarizes the basic visual paradigm of XQBE. A vertical line divides the *source* part of the query (on the left) from the *construct* part (on the right). Thus the query has a natural reading order from left to right. The source part describes the XML data to be matched against the set of input documents, while the construct part specifies which parts will be retained in the result, together with (optional) newly generated XML items.[2] Both parts can be annotated to express selection predicates,

[1]For the sake of brevity, we assume the reader is familiar with XQuery; a good example-driven introduction is W3C [2003a].

[2]The construct part also makes it possible to prune and project the XML fragments matched in the source part, thus supporting some additional querying capabilities, which are not exclusive of the source part. This may seem in contrast with the "query on the left, construct on the right" core message of the paradigm. However, it will be clarified that these prunings and projections are intrinsically related to the construction of the result.

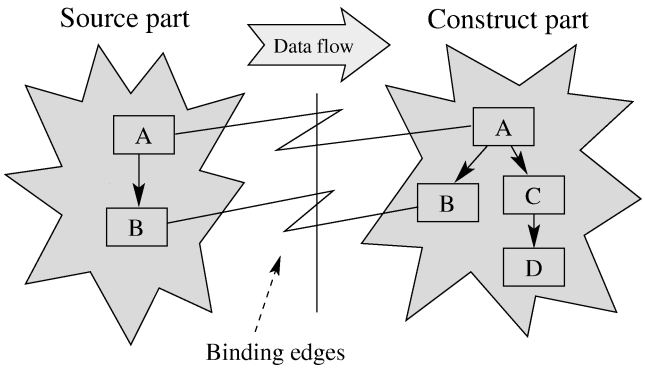


Fig. 1. The visual querying paradigm of XQBE.

and the correspondence between the components of the two parts is expressed by means of explicit binding edges.

1.1.2 *Expressive Power.* XQBE allows for arbitrarily deep nesting of XQuery FLWOR expressions, supports construction of new XML elements, and permits restructuring existing documents. Of course, XQBE is not equivalent to XQuery, which is Turing-complete (a proof can be found in Kepser [2002]). As an example, XQBE does not support user defined functions, as we believe that a user confident with this abstraction can directly use XQuery; also, XQBE does not support disjunction (this is typical of many visual interfaces). These limitations are precise design choices, since we believe that a complete but too complex graphical language would fail both in replacing the textual language and in addressing usability requirements.

The Table of Figure 2 indicates for every XQuery feature whether it is supported by XQBE or not; we also provide examples for all the supported features.

1.2 Related Work

Since the introduction of XML, several textual query languages were proposed and analyzed by the database community [Fernandez et al. 1999a; Ives and Lu 2000], far before the proposal of XQuery [W3C 2003b].

XQBE, in turn, comes after a long stream of research on graph-based logical languages, started many years ago with QBE [Zloof 1977], a user friendly query language in which the user can formulate simple queries by filling in skeleton tables with an example of possible answers. A relationally complete visual query language that supports recursion is QBD* [Angelaccio et al. 1990]. QBD* is characterized by a uniform graphical interface for both schema specification and query formulation, based on the use of an EntityRelationship oriented data model. Its main idea is to provide the users with a large set of graphical primitives, in order to extract in a friendly way, the required information from the database schema and deal uniformly with the same graphical environment during all the interaction with the database, without textual intermediate.

The first *object-oriented* graphical query languages were G [Cruz et al. 1987] and G+ [Cruz et al. 1988]. In turn, Graphlog [Consens and Mendelzon 1990]

feature	*XQuery*	*XQBE*	*exemplifying query*
Universal quantification	yes	no	
Existential quantification	yes	yes	q9 (2.2.5)
Conjunction	yes	yes	q10 (2.2.5)
Breadth projection	yes	yes	q3, q4 (2.2.2)
Depth projection	yes	yes	q21 (3.2.1)
Renaming	yes	yes	q3, q4 (2.2.2)
Filtering	yes	yes	q22 (3.2.2)
Disjunction	yes	no*	
Construction of new elements	yes	yes	q5, q6 (2.2.3)
Join	yes	yes	q14, q15 (2.2.8)
Multiple documents	yes	yes	q14 (2.2.8)
Cartesian product	yes	yes	q7, q8 (2.2.4)
Negation	yes	partially	q17 (3.1.1)
Union	yes	no	
Difference	yes	no*	
Aggregates	yes	yes	q18 (3.1.2)
Arithmetic computations	yes	yes	q19 (3.1.3)
Grouping**	no	no	
Sorting	yes	yes	q23 (3.2.3)
Querying schema order	yes	no	
Querying instance order	yes	no	
Nesting	yes	partially	q11, q12 (2.2.6)
Flattening	yes	yes	q7 (2.2.4)

* *XQBE has no explicit "or" and "minus" operators, but some form of disjunction and some form of difference are implicitly supported because of the partial support for negation.*

** *XQuery has no explicit grouping construct (such as the SQL group by). Implicit grouping can be performed by means of document restructuring; accordingly, some support to grouping in XQBE is achieved by means of restructuring as well.*

Fig. 2. Covered querying capabilities.

and Good [Paredaens et al. 1992] descend from G+; Good offers a uniform notation for object databases where nodes represent objects and edges represent relationships. A Good-like notation was used by G-Log [Paredaens et al. 1995], a logic-based graphical language that makes it possible to represent and query complex objects by means of directed labeled graphs. An evolution of this language, WG-Log [Comai et al. 1998], was built to query internet pages and semi-structured data adding to G-Log some hypermedia features. A direct descendent of WG-Log was XML-GL [Comai et al. 2001], an early, self-standing visual query language for XML, designed far before XQuery. XQBE can then be considered as an evolution of XML-GL, specifically targeted to be a suitable visual interface for XQuery.

Given this short "language history," a detailed comparison between XQBE and XML-GL, its main predecessor, is given in Appendix A. The next section includes a short description of other visual languages that have been proposed for XML and for generic semi-structured data.

1.2.1 *Other Visual Query Languages.* QSByE (Querying Semi-structured data By Example [Filha et al. 2001]) is a graphical interface that represents data as nested tables and extends the QBE paradigm to deal with semi-structured data. MiroWeb Tool [Bouganim et al. 1999] uses a visual paradigm based on

trees that implements XML-QL. QBEN is a graphical interface to query data according to the nested relational model; the users specify their queries with the operations of the nested relational algebra [Jaeschke and Schek 1982].

Equix [Cohen et al. 1999] is a form-based query language for XML repositories, based on a tree-like representation of the documents, automatically built from their DTDs. Equix supports the visual construction of complex queries including quantification, negation and aggregation; it has limited restructuring capabilities (the only restructuring primitive is the introduction of new nodes). In Cohen et al. [2000] a new syntax for Equix is proposed, which enhances the language's user-friendliness and is specialized for Web search.

BBQ [Munroe and Papakonstantinou 2000] (Blended Browsing and Querying) is a graphical user interface proposed for XMAS [Ludaescher et al. 1999], a query language for XML-based mediator systems (a simplification of XML-QL). In BBQ XML elements and attributes are shown in a directory-like tree and the users specify possible conditions and relationships (as joins) among elements.

PESTO [Carey et al. 1996] (Portable Explorer of STructured Objects) is an integrated user interface that supports browsing and querying of object databases; PESTO allows users to navigate in a hypertext-like fashion, following the relationships that exist among objects. In addition, it allows users to formulate object queries through a unique, integrated query paradigm that presents querying as a natural extension of browsing. PESTO includes support for basic query operations (such as simple selections, value based joins, universal quantification, negation, and complex predicates). VQBD [Chawathe et al. 2001] addresses the objective to explore an XML document of unknown structure.

XQForms [Petropoulos et al. 2001] is a generator of Web-based query forms and reports for XML data. XQForms takes as input the XML Schema, a declarative specification of the logic of the query, and a set of template libraries. The use of these three different inputs allows a clear separation between data to be queried, query logic, and presentation of the results.

QURSED [Papakonstantinou et al. 2002] allows the development of Web-based query forms and reports (QFRs) for XML data. QURSED is based on the QSS formalism, a capability-description language [Levy et al. 2002; Vassalos and Papakonstantinou 2000], and produces XQuery-compliant queries. QURSED allows the user to use both conjunction and disjunction; disjunctive queries are pre-processed, and OR conditions are substituted by a forest of condition trees without OR nodes, called *conjunctive condition trees*. The QURSED Editor inputs the XML Schema that describes the structure of XML data and an HTML query form page (that provides the visual part of the form page). The editor displays the XML Schema and the HTML pages to the developer, who uses them to visually build the query set specification and the query/visual association (that indicates how each parameter is associated to HTML form). Then a compiler generates Java Server Pages, which control the interaction with the end user.

Many other visual languages have been proposed for data management on the Web, including WebML [Bongio et al. 2001], Araneus [Atzeni et al. 1997] and Strudel [Fernandez et al. 1999b]. All of them offer a data model, a

navigation model, and a presentation model. They decouple the query aspects of Web development from the presentation ones, but support very simple query expressions that cannot be compared to the full XQuery or to the fraction of XQuery supported by XQBE.

1.3 Article Organization

XQBE is progressively introduced in Sections 2 and 3. Section 2 introduces the core constructs of XQBE, which are formally defined through the use of a graphic syntax and then exemplified with classic queries inspired by the W3C "XML Query Use Cases" [W3C 2003a]. Section 3 is dedicated to the advanced constructs of the language (negation, aggregates, computations, depth projection, conditional construction, and sorting). Then, Section 4 defines the semantics of the core constructs of XQBE by means of expressions that extract information from the source document and functions that recursively build the query result. Section 5 describes how XQuery expressions can be generated according to the semantics and presents our prototype implementation of XQBE. Section 6 concludes the article, while an appendix compares XQBE with XML-GL, its main predecessor.

2. CORE SYNTAX OF XQBE

In this section, we introduce *CoreXQBE*, a self standing subset of XQBE, which provides basic querying capabilities; CoreXQBE includes language constructs that need to be presented together. Section 3 will present additional constructs of XQBE that can be incrementally described as orthogonal additions to CoreXQBE; thus, we progressively present the full expressive power of the language.

2.1 Syntax of CoreXQBE

We first present the basic elements of the language, then give an overall, informal view of CoreXQBE queries, then describe the source and construct part, then discuss interconnecting edges.

2.1.1 *Basic Elements of XQBE.* The visual representation of XML documents in XQBE relies on a simplified XML data model, basically reduced to the notion of Elements, Attributes, and PCDATA content, with containment hierarchies connecting such elements. Data types as described in XML Schema are not supported, and ID/IDREF couples are treated just like all other attributes (thus, the underlying XML data model is even simpler than a DTD specification).

Queries in XQBE use a tree representation whose nodes represent the elements, attributes, and PCDATA content of given XML documents.

(a) *Element nodes* (*E-Nodes*) are shaped as labeled rectangles; their label represents the element name (or tagname).
(b) *PCDATA nodes* (*P-Nodes*) are represented as empty circles and denote the textual content of XML elements.

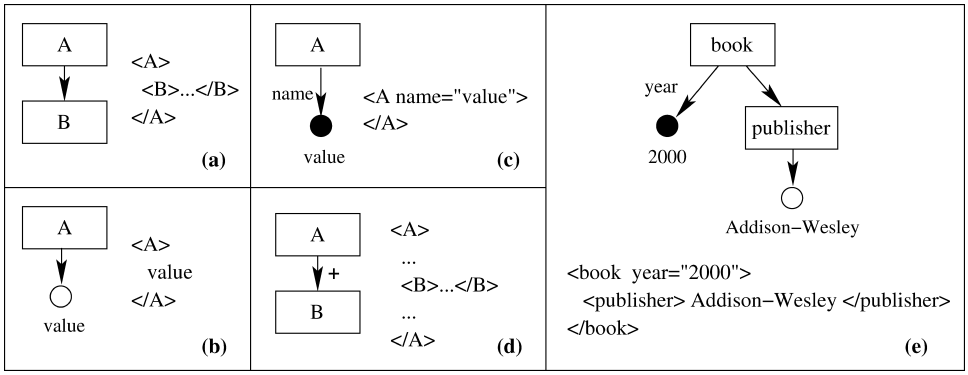


Fig. 3. Basic elements of XQBE.

(c) *Attribute nodes* (*A-Nodes*) are represented as filled (black) circles. The label on the incoming arc represents the attribute name.

(d) The *containment relationship* between two XML items is represented by means of a directed arc from the container to the contained item. Arcs labeled with a cross (that reminds one of the Kleene cross operator) express the ascendant-descendant relationship: the transitive closure of the relationship.

A-Nodes and P-Nodes together can be referred as *value nodes* (*V-Nodes*), as they represent the actual data content[3] of the XML documents. They may be labeled with constant values that enable the expression of matching predicates against the actual content of XML documents; equality is implicitly used as comparison predicate, but any other comparison predicate can be added to the label.

The aforementioned components combine into structures like those in Figure 3. The figure shows a direct containment (a), PCDATA content (b), attributes (c), and the transitive closure of the containment relationship (d). At the right of the graphic representation, to help the reader's intuition, we present an XML document that "matches" each structure, interpreted as constructs of the source part of an XQBE query. To summarize the concepts introduced so far, Figure 3(e) represents an XQBE query matching with book elements having as year of publication "2000" and as publisher "Addison-Wesley;" the former condition is built upon an attribute, the latter upon the PCDATA content of the publisher element.

[3]XQBE has no graphical constructs to represent different parts of the textual content of an element, nor parts of the textual value of an attribute. Such values are represented and referenced as a whole by one empty or filled circle respectively, and this is also true both in the case of the PCDATA content of the several text chunks of a mixed element (i.e. an element whose content is an arbitrary intermixture of PCDATA and subelements) and in the case of an attribute whose content represents multiple values, as for attributes of type IDREFS. In the case of mixed elements, the value represented by a P-Node is the concatenation of all the text excerpts at the first level of nesting (a deep concatenation can be expressed by means of a Kleene cross on the arc going from the E-Node to the P-Node). This inability to distinguish substrings is a precise design choice, motivated by the need for simplicity; we believe that a visual construct intended to cut a string into substrings would be rather confusing, and of marginal importance for most users.

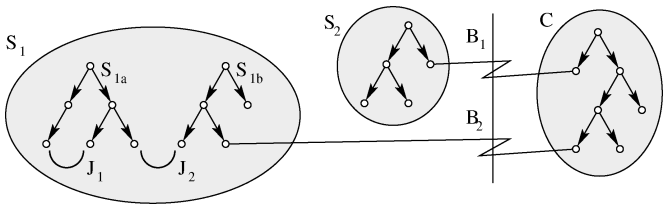


Fig. 4. How the tree-shaped components of a query can connect one to another.

2.1.2 *An Informal View of CoreXQBE.* As described in Figure 4, a query in CoreXQBE consists of two parts, called source and construct part. The source part (on the left) consists of one or more trees, possibly disjoint. Trees in the source part may be connected by inter-tree and intra-tree connections, representing joins that may occur within a document or between two documents. Join connections are the only allowed *confluences*: nodes with more than one incoming arc. These confluences are the only reason why the source part must be formally regarded as a collection of DAGs and not simply of trees. The construct part (on the right) has a single tree, denoting the query result. Binding edges connect the source and construct part; they are responsible for dictating which elements or values of the source documents should be part of the query result. Figure 4 shows an example of query with two join connections (J_1 is intra-tree, J_2 is inter-tree) and two binding edges (B_1 and B_2).

2.1.3 *Formal Definition of CoreXQBE.* We formally define CoreXQBE as the language of all queries q such that q is a triple $\langle S, C, B\rangle$, where S is the *source* graph of q, C is the *construct* graph of q, and B is a set of *binding edges* between the nodes of S and the nodes of C.

S is a sequence of connected DAGs d_i^S $(i = 1 \cdots n)$, each defined as a couple $\langle N_i^S, A_i^S\rangle$, where N_i^S and A_i^S are the sets of the nodes and arcs of d_i^S. We also define N^S as $\bigcup_{i=1}^n N_i^S$ and A^S as $\bigcup_{i=1}^n A_i^S$. The items contained in these two sets are named the source (or left) nodes and arcs of q.

C is a tree denoted as the couple $\langle N^C, A^C\rangle$, composed of the sets of the construct (or right) nodes and arcs of q.

B is a set of couples $\langle n_1, n_2\rangle$ where $n_1 \in N^S$ and $n_2 \in N^C$.

Last, we define $N = N^S \cup N^C$ and $A = A^S \cup A^C$ as the nodes and arcs of q.

Figure 5, summarizes the visual constructs of CoreXQBE and also exemplifies their use in some typical configurations. Each item introduces a node and discusses the constraints that apply to the nodes in various configurations.

2.1.4 *Source Part of CoreXQBE.* The **source part** S is used for expressing conditions on existing documents. S has the following elements:

(1) *E-Nodes* represent the XML elements of the source documents. Their labels cannot be omitted, but can be partially specified, as the wildcards '?' and '*' denote any character and any sequence of characters respectively (so that the most general specification is a '*' label, to denote *any* tagname).

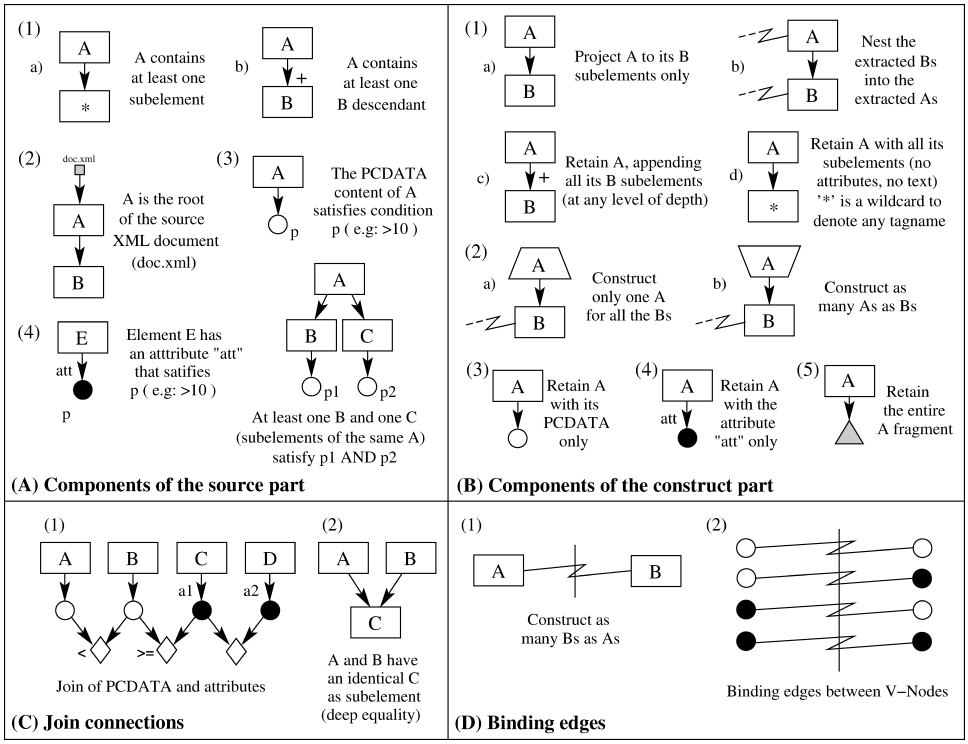


Fig. 5. The set of the graphical constructs of CoreXQBE and some characteristic configurations in the source (A) and construct parts (B), join connections (C), and binding edges (D).

(2) One *R-Node* (root node) is associated with each distinct document in the source part. Root nodes are represented as gray squares, labeled with the *location* of the corresponding XML document (typically its URI, see case A.2 in Figure 5).

(3, 4) *P-Nodes* and *A-Nodes* are used to express selection predicates that apply to the represented values; the label is composed of a comparison operator (=, >, ...) followed by a constant; equality is assumed as default if no operator is specified.

2.1.5 *Construct Part of CoreXQBE.* The **construct part** C is used to build the query result, by means of projections of the nodes extracted in the source part, possibly interleaved with newly generated items. N^C has the following node types:

(1) *E-Nodes* represent those elements of the source documents that are retained in the query result; every E-Node is either put by the query into a one-to-one correspondence with a node belonging to the source part—by means of a binding edge—or represents a projection of such a node—if the correspondence is imposed by an ancestor E-Node with a binding edge. Labels of

E-Nodes in the construct part cannot be omitted;[4] also, E-Nodes connected by a binding edge must include no wildcards. Given that E-Nodes depend on the source part, either directly or indirectly, a structural constraint applies: an E-Node without an incoming binding edge must be descendant of another E-Node with an incoming binding edge.

(2) *Trapezoidal nodes* (*T-Nodes*) represent newly generated elements (tags) to be included in the result of the query and thus are allowed only in C. Two kinds of new tags are allowed, represented by two kinds of T-Nodes:
   —*Single-tag* T-Nodes—trapezoidal nodes with the shorter edge on the bottom—denoting a new tag enclosing each instance of the concepts placed immediately below the node. Note that if the node has multiple successor nodes, then its effect is to build their Cartesian product by generating a new tag for each distinct combination of successor node instances.
   —*Set-tag* T-Nodes—trapezoidal nodes with the shorter edge on the top—denoting a unique new tag enclosing all the instances of the concepts placed immediately below the node. Note that if the node has multiple successor nodes, then its effect is to list all the instances of the first one, followed by all the instances of the second one, and so on.[5]

   It is possible to include in C subtrees only made of trapezoidal nodes, representing fragments that are generated in the result, but in order to avoid cumbersome configurations (that would be difficult to understand and use) we pose two further restrictions: at most one node in any of such subtrees can be a single-tag T-Node (so as to avoid Cartesian products of Cartesian products) and all the set-tag T-Nodes that are descendants of a single-tag T-Node in the subtree can have at most one outgoing arc (so as to avoid heterogeneous components within Cartesian products).

(3) *P-Nodes* represent PCDATA content of the elements included in the result. If a P-Node is labeled, the content is the constant value expressed by the label. If a P-Node is not labeled, either the content is computed as a projection of the E-Node under which the P-Node is placed, or the content is derived from the source part (if the node is connected by a binding edge).

(4) *A-Nodes* represent attributes included in the result. If an A-Node is labeled, its value is the constant value expressed by the label, which is assigned to the attribute whose name appears along the attribute edge. If an A-Node is not labeled, the attribute value is either extracted from an attribute of the element corresponding to the E-Node under which the node is placed, or derived from the source part (through a binding edge). All A-Nodes descending from the same E-Node in the construct part must have different labels, because in XML attribute names of a given element are unique.

[4]An allowed exception regards E-Nodes connected by a binding edge. If such nodes are not labeled, the label is determined by the connected node in N^S.

[5]The reader is warned that *single-tag* nodes indeed generate multiple tags in the result document, while *set-tag* nodes generate only one tag, which includes a set of successor nodes.

(5) *Fragment nodes* (*F-Nodes*) denote the inclusion of entire fragment: a gray triangle placed below an E-Node states that the corresponding element has to be included in the result with all its content (subelements, PCDATA, attributes).

The left-to-right order of nodes in N^C is used in constructing the query result.

2.1.6 *Join Connections and Binding Edges.* We now describe in detail the **join** connections (**C**) between nodes in N^S and the **binding edges** (**D**) between S and C.

(**C**) *Join connections* express either the *value-based* comparison between atomic values (PCDATA content or attribute values), or the *object-based* "deep" equality of two XML fragments.
- —In the case of value-based joins, the join connection is visually represented as a confluence into a *J-Node*: a rhomboidal node with *two* (or more) incoming arcs (as in Figure 5C, case 1). J-Nodes may be associated with a label to denote the join predicate; this label is optional, since an equi-join is assumed as default. Confluent arcs on a J-Node can only originate from V-Nodes.
- —In the case of object-valued equality, the join condition is visually represented as confluence into an *E-Node* (as in Figure 5C, case 2). Confluent arcs on an E-Node can only originate from E-Nodes. Deep equality means that the entire fragments being joined must be identical.

(**D**) The *binding edges* in B are represented as piecewise-linear curves that connect a node in N^S to a node in N^C, thus crossing the vertical line that separates the source and construct parts of the query. Binding edges can only connect E-Nodes to other E-Nodes and V-Nodes to other V-Nodes, as shown in Figure 5D. A node in N^C may be bound by at most one edge, because the item to be included is defined once in the source part; instead, the nodes of N^S may be bound by more than one edge, because the same item may be used several times in the constructions of C.

2.2 Examples of CoreXQBE

The set of visual constructs introduced so far allows one to formulate a large variety of queries to select, project and restructure XML data; progressive examples are shown next, targeted to the document of Figure 6 and inspired by the *XML Query Use Cases* published by the W3C [2003a]). We first focus on the construct part, then on the source part, then on binding edges, and finally on join conditions.

2.2.1 *The First Queries.* Consider the query q1 "*Return all books in the source document*." In XQuery:

```
for $b in doc("www.bn.com/bib.xml")/bib/book                    (q1)
return $b
```

```
<bib>

    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author> <last>Stevens</last> <first>W.</first> </author>
        <publisher>Addison-Wesley</publisher>
        <price> 65.95</price>
    </book>

    <book year="1992">
        <title>Advanced Programming in the Unix environment</title>
        <author> <last>Stevens</last> <first>W.</first> </author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>

    <book year="2000">
        <title>Data on the Web</title>
        <author> <last>Abiteboul</last> <first>Serge</first> </author>
        <author>  <last>Buneman</last>  <first>Peter</first> </author>
        <author>   <last>Suciu</last>    <first>Dan</first>  </author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>39.95</price>
    </book>

    <book year="1999">
        <title>The Economics of Technology and Content for Digital TV</title>
        <editor>
            <last>Gerbarg</last> <first>Darcy</first>
            <affiliation>CITI</affiliation>
        </editor>
        <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>

</bib>
```

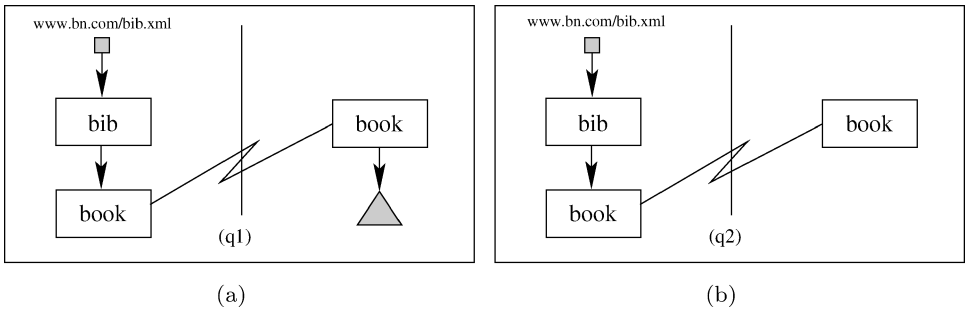
```
<!ELEMENT bib    (book*)>                          <!ELEMENT title       (#PCDATA)>
<!ELEMENT book   ( title, ( author+ | editor+ ),   <!ELEMENT last        (#PCDATA)>
                   publisher, price )>             <!ELEMENT first       (#PCDATA)>
<!ATTLIST book year CDATA #REQUIRED>               <!ELEMENT affiliation (#PCDATA)>
<!ELEMENT author (last, first)>                    <!ELEMENT publisher   (#PCDATA)>
<!ELEMENT editor (last, first, affiliation)>       <!ELEMENT price       (#PCDATA)>
```

Fig. 6. A sample document (http://www.bn.com/bib.xml) and its DTD.

Its XQBE version is in Figure 7(a). Data is extracted from the document `bib.xml` at the location `www.bn.com`, matching all the `book` elements that are contained into a `bib` element; the target document is denoted by an R-Node (the small gray square). In the construct part, the binding edge between the `book` nodes states that the query result shall contain as many `book` elements as those matched in the source part. Fragments below the extracted `book` elements are entirely retained in the result of the query; this is denoted in XQBE by means of a gray triangle (F-Node) below the bound node, to indicate the inclusion of all subelements at an arbitrary level of depth.

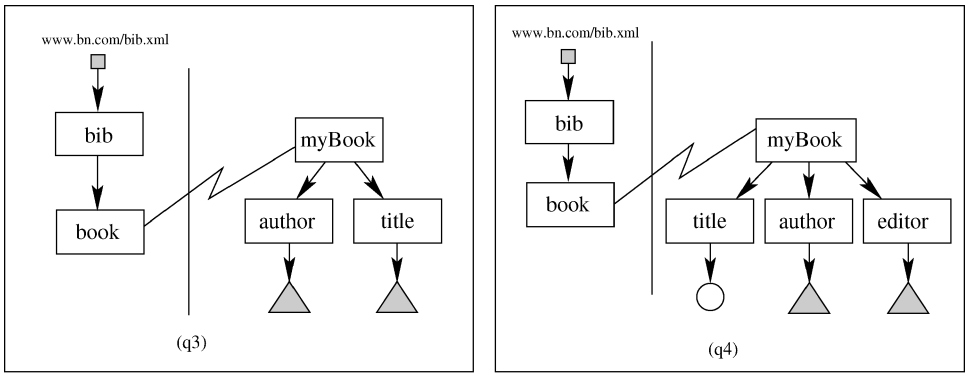
Consider now query q2, described in Figure 2(b), as the variant of query q1 in which the `<book>` tags are left empty. In XQuery:

```
for $b in doc("www.bn.com/bib.xml")/bib/book                    (q2)
return <book/>
```



(a) (b)

Fig. 7. Simple queries.



(a) (b)

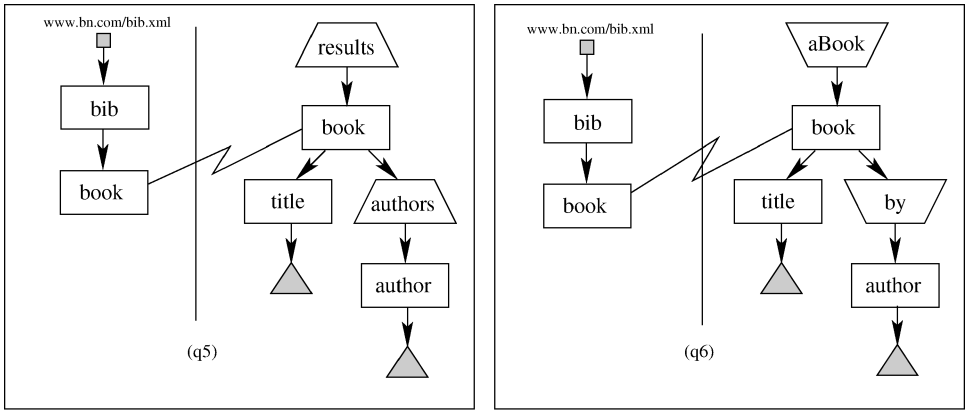
Fig. 8. Queries for projection and renaming.

The above variants represent two extremes in the construction of the result, one that retains an entire fragment below a retrieved element and one that prunes all its content. Any combination of *projections* is allowed between these two extremes, as shown in the next two examples.

2.2.2 *Element Projection and Renaming.* Consider the query q3 "*Return all books in the source document, retaining for each book only the list of their authors and the title; also change the tagname to myBook*". In XQuery:

```
for $b in doc("www.bn.com/bib.xml")/bib/book
return <myBook>                                                    (q3)
         { $b/author }
         { $b/title }
       </myBook>
```

This query shows how to express the renaming and projection of all the `book` elements. In the XQBE version of q3, shown in Figure 8(a), the binding edge between the `book` element and the `myBook` element causes the construction of a `myBook` element for each `book`. The `author` and `title` elements below `myBook` extract the corresponding subelements of `book`, thus projecting the `book` element.



(a) (b)

Fig. 9. Queries with new tags in the result.

It is important to note that the binding edge establishes an environment that enables one to consider `author` and `title` as subelements of `book`, even if `book` is renamed.

Another example of projection is given in q4, which projects the extracted books including their editors. The `title` is projected to its PCDATA content only, and the order of authors and titles is inverted with respect to q3.[6]

2.2.3 *Newly Generated Elements.* Query q5 is a variant of q3 stating "*For each book in the bibliography, list the title and authors, grouping all the extracted books inside a new* `results` *element and the list of authors of each book inside a new* `authors` *element*". In XQuery:

```
<results>
 { for $b in doc("www.bn.com/bib.xml")/bib/book                    (q5)
   return <book>
            { $b/title }
            <authors> { $b/author } </authors>
          </book>
 }
</results>
```

The XQBE version of Figure 9(a) shows the use of *set-tag* T-Nodes. The `results` node above the `book` node means that all the generated `book`s are to be contained into a single `results` element. This node represents a newly generated element. Similarly, one `authors` tag is generated for all authors of each book. Note that the newly generated tag does not change the context of element names, therefore

[6]By inspecting the DTD of the document in the running example (in Figure 6), we realize that `title` as element has only its PCDATA content, thus the two queries extract the same information for `title`. We can as well realize that books have either authors or editors but not both. Thus, for each book, one of the two branches will evaluate to an empty set. This data constraint cannot be captured by query languages.

author in the construct part is considered as a subelement of book in the source part.

Consider now, as a variant of q5, query q6, in which we like to add a new tag aBook around each extracted book and to add a new tag by around each extracted author. In this case we need to generate as many tags as there are books or authors, respectively. In XQuery:

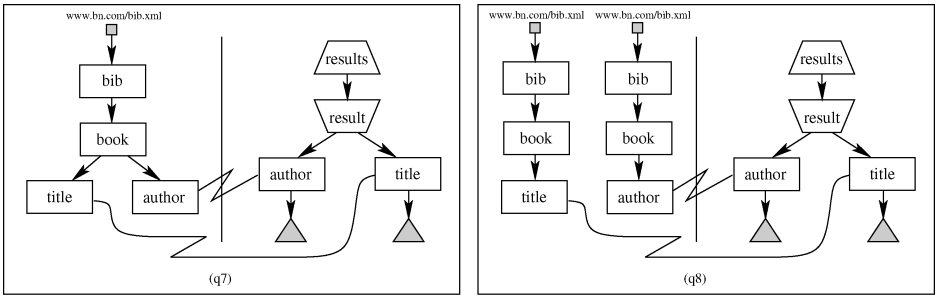
```
for $b in doc("www.bn.com/bib.xml")/bib/book
return <aBook>                                                    (q6)
         <book>
           { $b/title }
           { for $a in $b/author
             return <by> $a </by> }
         </book>
       </aBook>
```

The XQBE version of Figure 9(b) shows the use of *single-tag* T-Nodes. The aBook tag is replicated for each book node, and similarly the by tag is replicated for each author. Again, the newly generated tag does not change the context of element names, therefore author in the construct part is considered as a subelement of book in the source part.

2.2.4 *Flattening and Cartesian Product.* XQBE allows one to express many kinds of document transformations. As an example, we show how to synthetically express the flattening of hierarchical data structures. Consider the query q7 (Q2 in W3C [2003a]) "*Create a flat list of all the title-author pairs, with each pair enclosed in a* result *element*." In XQuery:

```
<results>
 { for $b in doc("www.bn.com/bib.xml")/bib/book,                  (q7)
       $a in $b/author
       $t in $b/title,
   return <result>
            { $a }
            { $t }
          </result>
 }
</results>
```

Its XQBE version is in Figure 10(a). The results element in the construct part is a new tag enclosing the result document; then, several result elements are generated by means of a *single-tag* T-Node; the cardinality of these elements is determined by the number of book elements that are extracted by the source part of the query. Each such element has a pair of successors tagged title and author, and such pairs (with all the fragments underlying each of them) are enclosed within each generated result tag. Note that the book element in the source part provides a common context, enabling the pairing of authors and titles only when they have actually written a book.



(a) (b)

Fig. 10. Queries showing flattening and Cartesian product.

Let us next consider a (rather unusual) query with two unrelated binding edges. Consider query q8 asking to build the Cartesian product of all possible pairs of authors and titles, regardless of the fact that the authors have indeed written a book. In XQuery:

```
<results>
 { for $a in doc("www.bn.com/bib.xml")/bib/book/author                    (q8)
       $t in doc("www.bn.com/bib.xml")/bib/book/title,
   return <result>
            { $a }
            { $t }
          </result>
 }
</results>
```

Its XQBE version is in Figure 10(b). Also in this case, several `result` elements are generated by means of a single-tag T-Node, but the cardinality of these elements is determined by the product of the number of different `title` and `author` elements, which are retrieved from two independent copies of the bibliography document. Thus, this query builds the Cartesian product of all titles and authors appearing in the source document.

2.2.5 *Queries with Existential and Selection Predicates.* With queries q1–q8, we have shown XQBE queries whose construct part is increasingly complex; we now focus on the source part. Consider the query q9 "*Return all books having an editor*." In XQuery:

```
<bib>
 { for $b in doc("www.bn.com/bib.xml")//book                              (q9)
   where exists($b/editor)
   return $b
 }
</bib>
```

The corresponding query in XQBE, shown in Figure 11(a), includes an existential quantification on the source part: in order for a `book` element to be

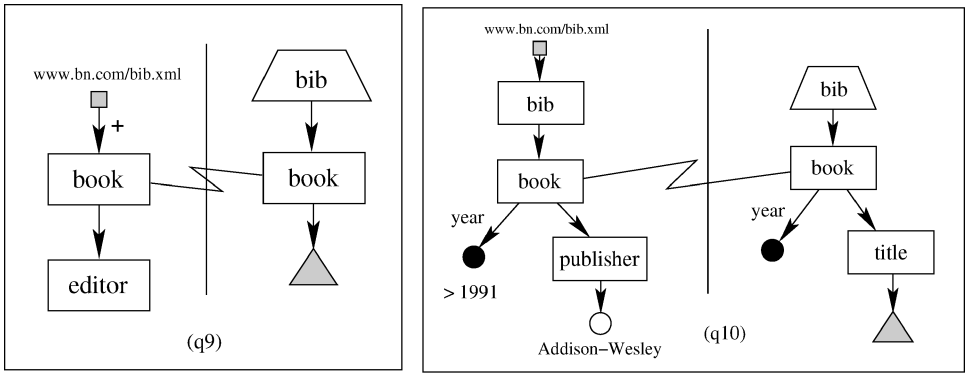


Fig. 11. Existential and select-project queries.

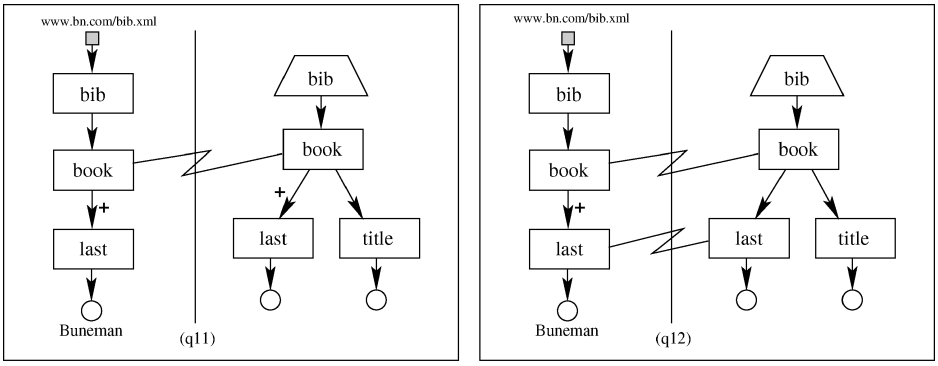
included, an `editor` element must exist. Note that in this query the `book` node is connected to the root node, which denotes the XML document by means of an arc, which is labeled with a '+', to state that `book` is not required to be the root element of the target document; instead, any `book` element at any level of nesting is considered.

Let us now consider a full selection query, with conjunctive predicates. Consider the query q10 (Q1 in W3C [2003a]): "*list books published by Addison-Wesley after 1991, including their year and title*." In XQuery:

```
<bib>
 { for $b in doc("www.bn.com/bib.xml")/bib/book                    (q10)
   where $b/publisher = "Addison-Wesley"
         and $b/@year > 1991
   return <book year="{$b/@year}">
            { $b/title }
          </book>
 }
</bib>
```

The XQBE version of q10 is in Figure 11(b). In this query, the source part matches all the `book` elements descending from a `bib` element that have a `year` attribute with a value greater than 1991 and contain a `publisher` element whose PCDATA content equals "Addison-Wesley." Note that XQBE supports conjunctive predicates but it does not support disjunctive predicates. In the construct part, the bound `book` nodes are projected upon their title and publication year of the selected books.

2.2.6 *Queries with Hierarchical Bindings.* We now focus on the binding passing mechanism of XQBE. Consider first the query q11 (Figure 12(a)) retrieving all books having an author or editor whose last name is "Buneman."



(a) (b)

Fig. 12. Selection with single and double binding edge.

The XQuery version is:

```
<bib>
 { for $b in doc("www.bn.com/bib.xml")/bib/book                    (q11)
   where $b//last = "Buneman"
   return <book>
            { $b//last }
            { $b/title }
          </book>
 }
</bib>
```

The XQBE version of query q11 is shown in Figure 12(a). Note that the selection predicate is used to filter all books satisfying the condition (i.e. such that at least one of their authors or editors has the last name equal to Buneman), but the entire set of last names is retained in the query result (the query result also contains the elements corresponding to Buneman's coauthors or coeditors).

Consider now query q12 as a further restriction of q11, which includes in the result only the last names satisfying the condition. The XQuery version is:

```
<bib>
 { for $b in doc("www.bn.com/bib.xml")/bib/book                    (q12)
   where $b//last = "Buneman"
   return <book>
            { for $l in $b//last
              where $l = "Buneman"
              return $l }
            { $b/title }
          </book>
 }
</bib>
```

This is obtained by means of hierarchical bindings. Note that not only the `book` element is bound in the construct part, but also the `last` element, which is

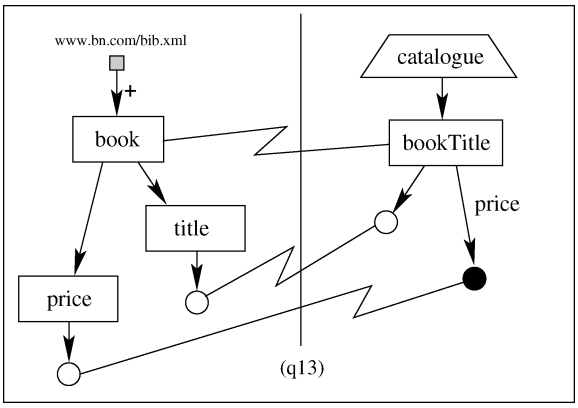


Fig. 13. Binding edges between V-Nodes.

hierarchically placed below `book`; hence, only the specific `last`, which descends from a given, bound book element in the source part is bound to the `last` element below such book in the construct part; in this way, the result contains only the element extracted by the source part condition. Note that the arc from `book` to `last` in the construct part is not labeled with '+', as it was in q11, because in the former case `last` elements had to be found by projecting the `book` node, while in the latter case the `last` elements are provided by the evaluation of the lower binding edge.

2.2.7 *Binding Edges that Transport Values.* Query q13 demonstrates how to transport atomic values from the source part to the construct part. It states "*List in a 'catalogue' tag a 'booktitle' tag for each book in the bibliography, with the title as PCDATA content and the price as an attribute*." In XQuery:

```
<catalogue>
 { for $b in doc("www.bn.com/bib.xml")//book                              (q13)
   where exists($b/price/text()) and exists($b/title/text())
   return <bookTitle price="{ $b/price/text() }">
            { $b/title/text() }
          </bookTitle>
 }
</catalogue>
```

The XQBE version of q13 is shown in Figure 13. Note that one binding edge connects two P-Nodes: this states that the PCDATA value of the new `bookTitle` element is the value as taken from the corresponding PCDATA in this source part. The other binding edge connects a PCDATA and an attribute: in this case the attribute value is taken from the PCDATA of `price` while the attribute name is specified on the attribute arc.

2.2.8 *Join Queries.* We turn now to join queries, and consider first the join between two documents (inter-document join). Consider the query q14 (Q5 in W3C [2003a]), which constructs a joint book catalogue, collecting information

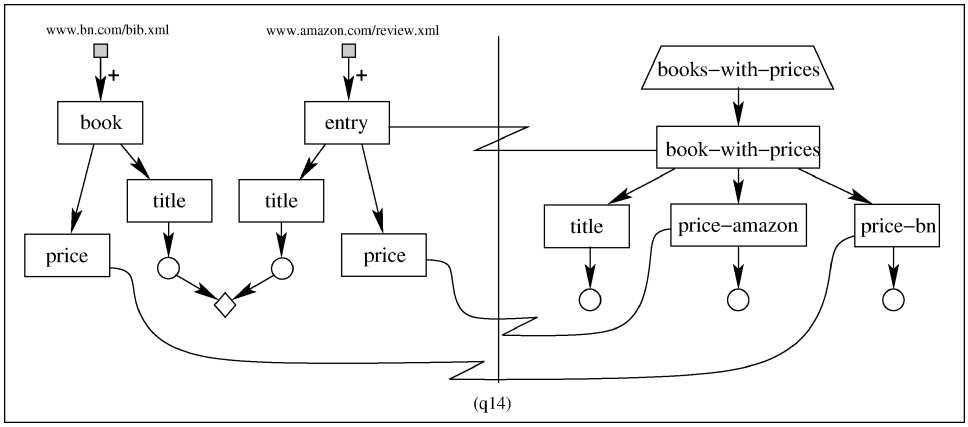


Fig. 14. Inter-document join.

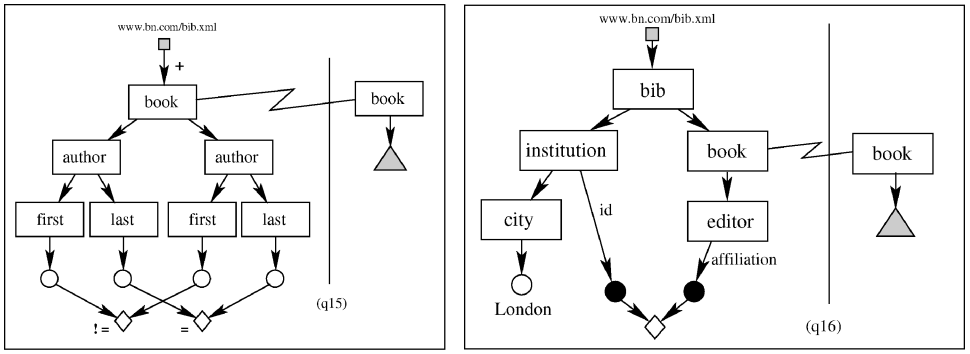
from different documents. It states "*For each book found at both bn.com and amazon.com, list the title of the book and its price from each source*." In XQuery:

```
<books-with-prices>
 { for $b in doc("www.bn.com/bib.xml")//book,                              (q14)
       $a in doc("www.amazon.com/review.xml")//entry
   where $b/title = $a/title
   return <book-with-prices>
            { $b/title }
            <price-amazon> { $a/price/text() } </price-amazon>
            <price-bn>     { $b/price/text() } </price-bn>
          </book-with-prices>
 }
</books-with-prices>
```

The XQBE query corresponding to q14 is shown in Figure 14. Value-based equality is expressed by means of a join connection between the PCDATA of the `title` elements. Then, for each `entry` in the `amazon` catalogue that gets bound after the join in the source part, its price and the price of the corresponding book in the `bn` catalogue are extracted from the two documents. Of course a symmetric solution is possible, that binds the `book` elements instead. As in query q5, the `books-with-prices` element in the construct part is a new tag enclosing the result document.

We consider next intra-document joins. Consider the query q15 extracting those books having two authors with the same lastname but different first name. In XQuery:

```
 for $b in doc("www.bn.com/bib.xml")/bib/book
 where some $a1 in $b/author satisfies                                     (q15)
       some $a2 in $b/author satisfies
       ( $a1/last = $a2/last and $a1/first != $a2/first )
 return $b
```



(a) (b)

Fig. 15. Intra-document joins.

The XQBE version of q15, shown in Figure 15(a), is based on the conjunction of two joins, having as arguments the PCDATA content of the `first` and the `last` elements, where elements of the first pair must be different, and elements of the second pair must be equal. All typical binary comparison predicates are allowed (<, <=, ! =, >, >=, =); equality is assumed as default if the predicate is unspecified. The *contains* predicate can be used between two PCDATA values to test if the second argument is a substring of the first argument.

Joins also express the equality of ID/IDREF pairs; in order to show an example of this kind of join, we assume a different DTD of the document. Let us add to editors the `affiliation` attribute (of type IDREF), referencing the `id` attribute (of type ID) of a new `institution` element, inserted as subelements of the `bib` root of the document:

```
<!ELEMENT bib (book+, institution+)>
<!ELEMENT editor (last, first)>
<!ATTLIST editor affiliation ID #REQUIRED>
<!ELEMENT institution (name, city)>
<!ATTLIST institution id ID #REQUIRED>
<!ELEMENT city (#PCDATA)>
<!ELEMENT name (#PCDATA)>
```

With this new DTD, we can express query q16, extracting "*books whose editor belongs to an institution located in London*." In XQuery:

```
for $bi in doc("www.bn.com/bib.xml")/bib,
    $bo in $bi/book                                          (q16)
where some $in in $bi/institution satisfies
      ( $in/city/text() = "London" and
        $in/@id = $bo/editor/@affiliation )
return $b
```

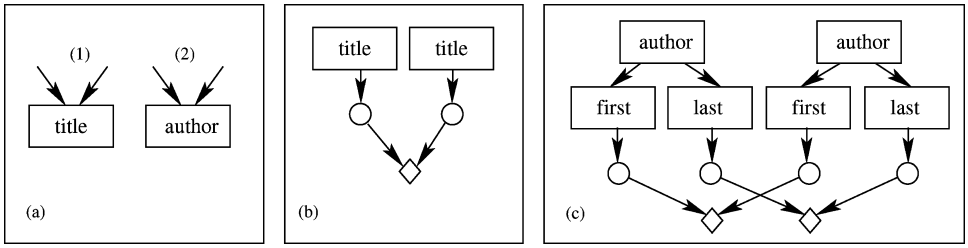


Fig. 16. Value-based joins and deep equality.

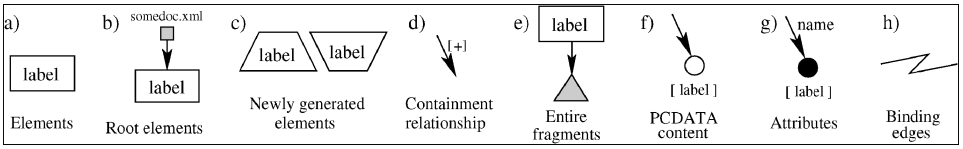


Fig. 17. The set of the graphical elements of CoreXQBE.

The ID/IDREF correspondence is treated in XQBE as a regular join on attribute values,[7] as exemplified by query q16 (in Figure 15(b)), where the `affiliation` and `id` attributes are equi-joined.

Joins seen so far are value-based, as they compare PCDATA or attribute values; Figure 16(a) shows join connections that express the deep equality—confluences—on the same E-Node. Such join condition is satisfied when at least one fragment contained in the left structure is equal to at least one fragment contained in the right structure. Note that, relative to the proposed DTD, the confluence on a `title` element (a1) is equivalent to the join upon PCDATA values (b), and similarly the confluence on the `author` element (a2) is equivalent to the conjunction of two joins upon the `first` and `last` PCDATA values (c).

2.3 Summary of the CoreXQBE Constructs

A summary of the constructs of CoreXQBE (in Figure 17) concludes this section.

3. ADVANCED XQBE CONSTRUCTS

We now complete the syntax of XQBE with advanced constructs, describing them by means of some examples.

3.1 Advanced Constructs in the Source Part

We first describe those advanced constructs allowed in the source part.

3.1.1 *Negation.* Negation nodes and arcs (*N-Nodes, N-Arcs*) are represented in XQBE by means of dashed figures (nodes and lines). They represent conditions that *must not* hold. Negated elements (*NE-Nodes*) and PCDATA nodes (*NP-Nodes*) are dashed as in Figures 18(S, case 1) and 23(a). All arcs

[7]The case of attributes of type IDREFS, i.e. lists of identifiers within the same attribute value, can be addressed with the `contains` join predicate, as XQBE does not allow one to extract parts of a value represented by V-Nodes (see the footnote in Section 2.1.1).

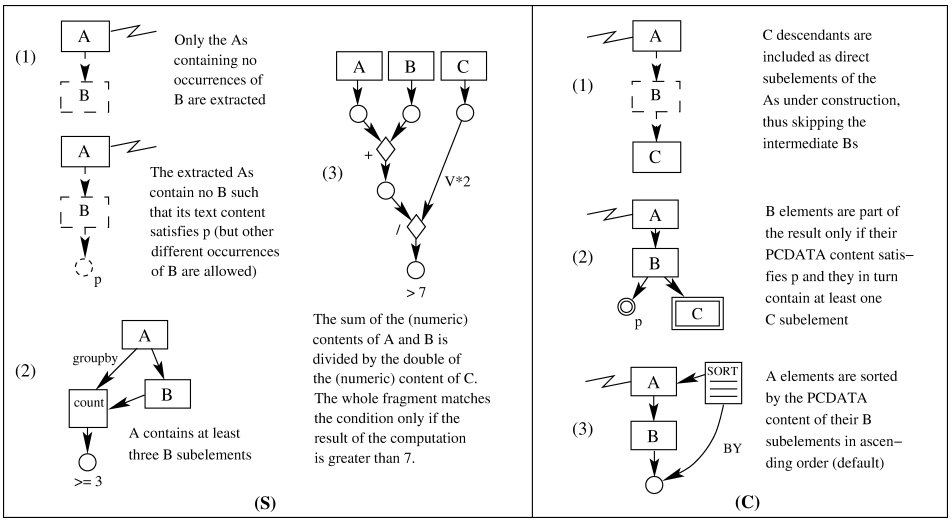


Fig. 18. The set of the advanced constructs of XQBE with some examples.

incoming to, or outgoing from, an N-Node must be dashed as well. Negated attribute nodes (*NA-Nodes*) are represented just like A-Nodes (filled black circles), but they can be distinguished because of the incoming dashed arc. NA-Nodes and NP-Nodes together are named *negated value nodes* (*NV-Nodes*). Negated nodes cannot be followed by "positive" (i.e. non negated) nodes and cannot be connected by binding edges.

The query q17 ("*List all the books not published by Addison-Wesley*") exemplifies the use of N-Nodes. It translates to the following XQuery statement:

```
<list>
 { for $b in doc("www.bn.com/bib.xml")/bib/book                    (q17)
   where not(some $p in $b/publisher/text() satisfies
             ( $p = "Addison-Wesley" ) ) )
   return $b
 }
</list>
```

In this example we ask for `book` elements inside which no `publisher` elements exist with a PCDATA content equal to "Addison-Wesley."

If no label were specified on the NP-Node, the global requirement would be more strict, discarding all books with a publisher (disregarding the PCDATA value). In the general case, as in the case of the positive predicates, the negative predicates must hold in conjunction (the fragment must satisfy the conjunction of the negation of the predicates).

3.1.2 *Aggregates.* Referring to Figures 18(S, case 2) and 23(b), Aggregate functions are represented by means of square nodes (*AG-Nodes*) labeled with the name of the function (min, max, count, avg, sum), with an incoming arc from the value subject to the aggregation and one optional incoming arc labeled `groupby`, from an element yielding the grouping context. Such context must be

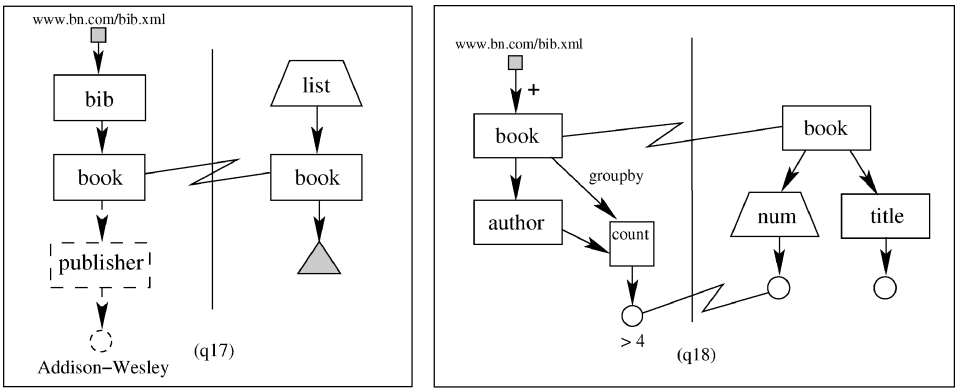


Fig. 19. Negation and aggregation nodes.

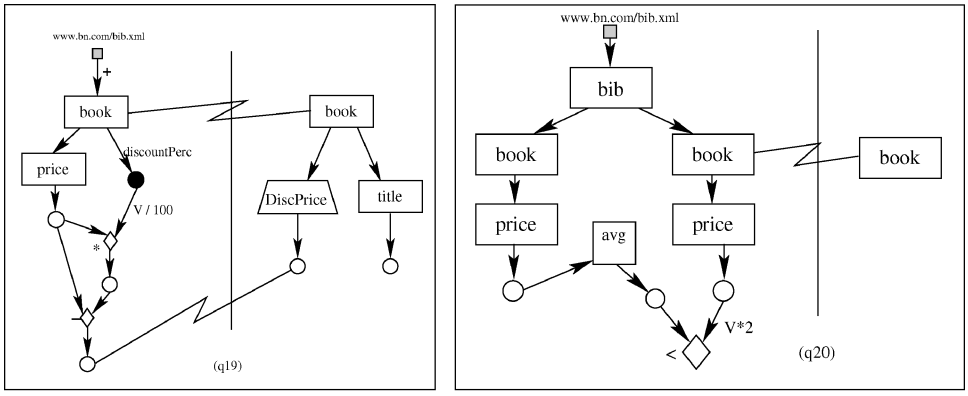
an element that hierarchically contains the value subject to aggregation. If the ingoing context is missing, the aggregate is computed over all the values bound by the query (yielding a scalar), else it is computed over all values sharing the same context. The computed (aggregate) value is represented by a V-Node reached by the outgoing arc, which may be subject to a predicate on the computed value. The computed V-nodes can be bound to nodes of the construct part.

The query q18 states "*List all the books with more than four authors, including their title and the number of authors*." In XQuery:

```
for $b in doc("www.bn.com/bib.xml")/bib/book
where count($b/author) > 4                                    (q18)
return <book>
         <num> { count($b/author) } </num>
         { $b/title }
       </boox>
```

The XQBE version is shown in Figure 19(b) and exemplifies the use of AG-Nodes.

3.1.3 *Arithmetic Computations.* Computation nodes (*C-Nodes*) are represented as small rhombuses, in which the incoming arcs represent the operands for the computation and the operator (+, *, -, /) is expressed by a label on the left of the node. Note that the use of rhombuses doesn't make the notation ambiguous with that of join nodes, because the sets of allowed labels are disjunct (comparators and operators). Arcs outgoing from a C-Node "transfer" the computed value to other nodes (typically V-Nodes) that may participate in further computations or comparisons. Moreover, all arcs adjacent to a C-Node may be labeled with expressions (such as 'V*2-14', where V represents the value of the node) that modify the transferred value. Thus, computations that apply to values represented by one node are specified by means of labels on the arcs that transport the values, while operations involving values from two nodes are specified by means of computational nodes.



(a) (b)

Fig. 20. Computation nodes (q19) and another aggregate (q20).

Query q19 in Figure 20(a) requires extending the DTD of the running example by adding a `DiscountPerc` attribute to `Book`s, whose value ranges from 0 to 100, and states "*Produce a list of books with a discount, including the discounted price (calculated by subtracting the discount from the price) and the title*." In XQuery:

```
for $b in doc("www.bn.com/bib.xml")//book
    $pr in $b/price,                                              (q19)
    $pe $b/@discountPerc
return <book>
         <DiscPrice> { $pr - ( $pr * ( $pe / 100 ) ) } </DiscPrice>
         { $b/title }
       </book>
```

Note that complex computations are represented as upside-down trees. Such reversed trees are isomorphic to the syntactic parse tree of the algebraic parenthesized expressions they represent (as shown in Figures 18(S, case 3) and 20(a)).

The query q20 in Figure 20(b) exemplifies the combined use of C-Nodes and AG-Nodes. The query states "*Find all the books whose price exceeds the double of the average price of the books in the catalogue*." In XQuery:

```
for $bi in doc("www.bn.com/bib.xml")/bib,
    $bo in $bi/book                                               (q20)
where avg( $bi/book/price ) < ( $bi/price * 2 )
return <book/>
```

3.2 Advanced Constructs in the Construct Part

Then we consider the advanced constructs only allowed in the construct part.

3.2.1 *Ghost Nodes.* Ghost nodes (*G-Nodes*) are represented by means of dashed rectangle nodes. They are used to explicitly mention an XML

element that will not be included in the constructed result but contributes to the construction.[8]

This feature is exemplified by the query q21 "*For each book list only the title and the surnames of the authors (maintaining the books in the order of the original document)*." In XQuery:

```
for $b in /bib/book
return <book>                                                    (q21)
         { $b/title }
         { $b/author/last }
       </book>
```

This is quite similar to the queries q3 and q4 that project the `book` elements "in breadth," but dealing with trees also requires the ability to project them "in depth" (i.e. to take far descendants of a given element and place them as direct subelements of that element, pruning the elements in the middle). In this case the `author` elements are pruned from the generated result, and `last` elements are directly inserted into the `book` elements.

3.2.2 *Conditional Construction.* Double-lined constructs, as in Figures 18 (C, case 2) and 23(d), represent conditional element nodes (*CE-Nodes*), conditional PCDATA nodes (*CP-Nodes*), and conditional attribute nodes (*CA-Nodes*), with the obvious associations to the shapes. The subgraphs composed of these nodes represent conditions that apply to the single-lined nodes they are attached to. These constructs are useful to specify conditions in order to prune XML items during the construction of the results, when such conditions cannot be expressed in the source part (because they would also impose an existential constraint). The following example clarifies this distinction.

Query q22 ("*Make a list of all the books with their title, including the editors only if they are affiliated to CITI*") is expressed in XQuery as follows:

```
for $b in doc("www.bn.com/bib.xml")/bib/book
return <book>                                                    (q22)
         { $b/title}
         { $b/editor[affiliation="CITI"] }
       </book>
```

This query imposes a constraint (the affiliation to CITI) that does not intervene in the selection of the matching source data, but only prunes the extracted XML items during the construction of the result. This is done in XQuery by placing filters into the path expressions of the `return` clause.

Note that the affiliation constraint cannot be specified in the *source* part (this would prune all the books without an editor from CITI), but has to be put in

[8]Therefore, dashed rectangles represent a negated condition when included in the source part and represent elements that should not be retained in the result when included in the construct part. Although these two meanings are different, the use of dashed nodes in the two parts of XQBE queries was found very intuitive and natural by XQBE users—the meaning of a dashed node in either part can be summarized as "this node is not in the document."

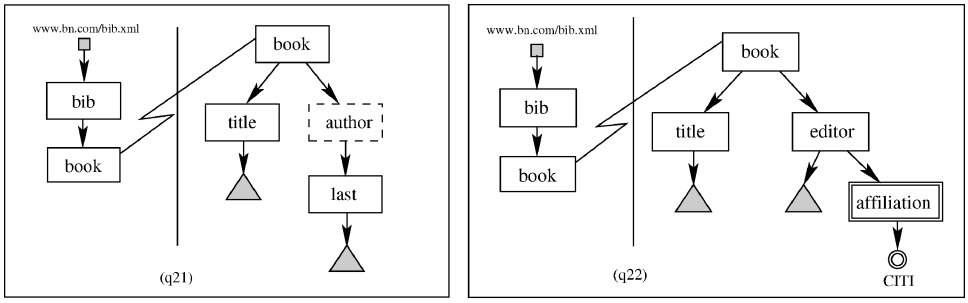


Fig. 21. Ghost nodes and conditional construction.

the *construct* part. In general a double-lined subtree applies a restriction only to the (single-lined) element in which it is rooted.

The XQBE version of the query is in Figure 21(b).

The newly introduced components require one to extend the notion of containment in the construct part, which must satisfy the following constraints:

(1) a positive node in C can descend only from a positive or ghost node, but not from a conditional node;
(2) ghost nodes must have at least a descendant (they are meaningless as leaf nodes, as they would represent path expressions that evaluate to nodes not to be included in the query result).

3.2.3 *Sorting.* Sorting is expressed by means of *S-Nodes*. The nodes to be sorted are reached by the (only) unlabeled outgoing arc, while the sorting criteria are the nodes reached by the "BY" arcs; the optional ASC or DESC keywords can be added as well, with ASC used as the default. If there is more then one ordering, an optional number between brackets indicates the ordering priority, and if no order is specified, the counterclockwise order from the unlabeled arc is assumed as default.

Consider the query q23 (also Q7 in W3C [2003a]) "*List books published by Addison-Wesley after 1991, with their year and title, sorting the retrieved books in lexicographic order*." This is a refinement of q10 (in Section 2.2.5), with the addition of a sorting criterion in the construction of the result. It translates to:

```
<bib>
 { for $b in doc("www.bn.com/bib.xml")/bib/book                    (q23)
   where $b/publisher="Addison-Wesley" and $b/@year>1991
   order by $b/title
   return <book>
            { $b/@year }
            { $b/title }
          </book>
 }
</bib>
```

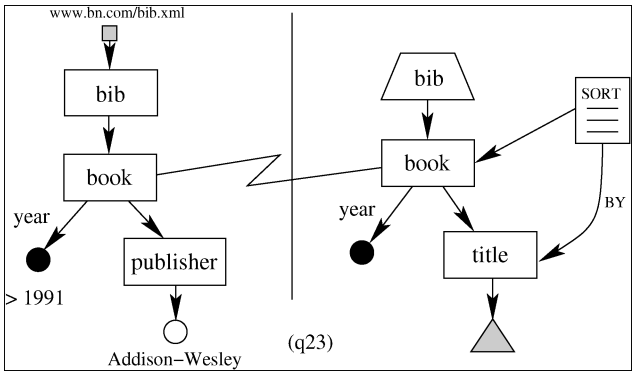


Fig. 22. Sorting.

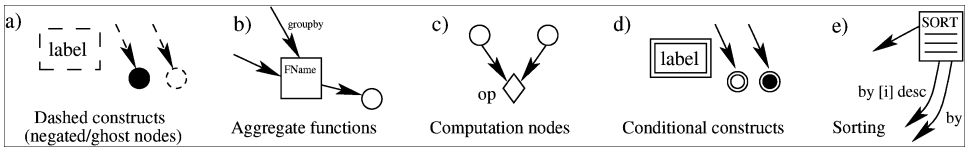


Fig. 23. The set of the advanced constructs of XQBE.

Note that the only difference between the XQuery versions of this query and q10 is the addition of the `order by` clause. Accordingly, in the graphical representation we just need to add an S-Node to the visual version of q10 (see Figure 22).

3.3 Summary of the Advanced Constructs

A summary of the advanced constructs of XQBE is in Figure 23.

4. FORMAL SEMANTICS OF COREXQBE

This section defines the formal semantics of CoreXQBE. We restrict the formalization to CoreXQBE, but indeed the generalization to the complete XQBE language is not difficult, once that the semantics of CoreXQBE is formally defined.

This section is organized as follows: Section 4.1 exploits the well known notion of tree pattern queries (TPQs) to denote the semantics of the source part by means of suitable TPQs, derived from the topology of the XQBE source graphs. Section 4.2 explains the order in which such TPQs should be computed, that depends on the construct part. Finally, Section 4.3 defines the construction of the query result in terms of a recursive traversal of the construct part.

NOTATION: we distinguish between **query nodes**, denoted by upper case letters, corresponding to the nodes of the query graph, and **instance nodes**, denoted by lower case letters, corresponding to actual nodes in the XML documents being extracted or constructed by the query.

4.1 Semantics of the Source Part

Tree pattern queries (*TPQs* [Amer-Yahia et al. 2001; Lakshmanan et al. 2004]), introduced to capture a significant fragment of XPath, are here adopted for modeling the extraction of the XML fragments that match the source part of a query.

The semantics of the source part is given by constructing a tree pattern query whose evaluation produces sets of instance nodes that are next used in the construct part.

4.1.1 *Tree Pattern Queries.* A TPQ is a triple $\langle \mathcal{V}, \mathcal{E}, \mathcal{F} \rangle$, where $\langle \mathcal{V}, \mathcal{E} \rangle$ is a rooted tree whose nodes $\mathcal{V}$ are labeled by variable names and whose edges $\mathcal{E}$ are partitioned into two sets ($\mathcal{E}_c$ and $\mathcal{E}_d$), respectively denoting the child and descendant axes of XPath. Edges in $\mathcal{E}_c$ and $\mathcal{E}_d$ are visually represented by means of single-lined and double-lined arcs respectively; they may also be denoted by means of predicates pc($\$N_1$, $\$N_2$) and ad($\$N_1$, $\$N_2$) respectively (where $\$N_1$,$\$N_2 \in \mathcal{V}$). $\mathcal{F}$ is a conjunctive formula, composed of tag constraints (TCs) and value-based constraints (VBCs)[9]:

a. TCs are of the form `$X.tag=T`, where `T` is a tag name. We will use TCs for constraining the tag names according to the node labels in the XQBE source graph. Note that the variables, names are in capitals, to recall that they are associated to query nodes.
b. VBCs are selection constraints comparing content values, attribute values, or constant values:

```
(($X.val | $X.attr1) comp (c | $Y.attr2 | $Y.val))
```

where `comp` $\in \{ =, \neq, <, \leq, >, \geq \}$, `attr1`, `attr2` represent attributes, `val` represents content, and `c` is a constant.

Answers for TPQs are formalized using matchings. A matching of a TPQ Q to an XML document collection C is a function m that maps query nodes of Q to instance nodes of C such that:

—structural relationships are preserved—for all $\langle \$X, \$Y \rangle \in \mathcal{E}_c$ $m(\$Y)$ is a child of $m(\$X)$ in C and for all $\langle \$X, \$Y \rangle \in \mathcal{E}_d$ there is a path from $m(\$X)$ to $m(\$Y)$ in C; and
—the formula $\mathcal{F}$ is satisfied.

A matching for Q provides a *set of bindings*, each binding being a n-uple of instance nodes that all together, orderly assigned to the variables $\$V_i \in \mathcal{V}$, satisfy the formula $\mathcal{F}$. We say that the set of bindings is the result of the *evaluation* of Q.

4.1.2 *Correspondence Between Source Graphs and TPQs.* The source part of an XQBE query graph can always be put into a correspondence with a TPQ,

[9]In addition, TPQs support predicates between pairs of nodes for expressing node identity constraints (NICs), but we do not use such a feature for representing source parts of queries.

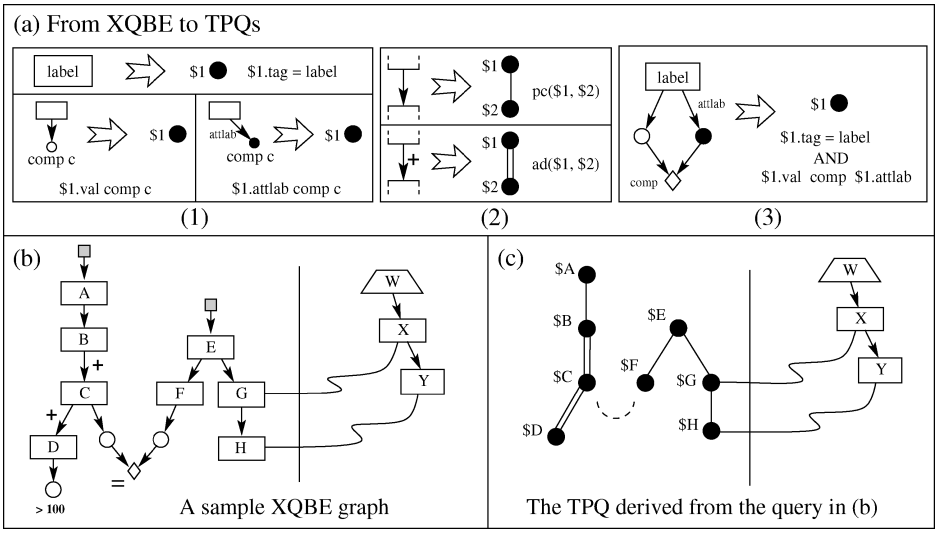


Fig. 24. Mapping XQBE graphs to TPQs.

built according to three principles, visually summarized in Figure 24a:

a1. Each distinct E-Node maps to one node in $\mathcal{V}$. Labels of E-Nodes map to TCs. Conditions on P-Nodes and A-Nodes map to VBCs that compare attribute values and element contents with a constant.

a2. Containment arcs map to $\mathcal{E}_c$ edges; arcs labeled with '+' map to $\mathcal{E}_d$ edges.

a3. J-Nodes map to VBCs that compare two attribute values or element contents.

The formula $\mathcal{F}$ is the conjunction of the aforementioned constraints.

As an example of the construction of a TPQ from an XQBE graph we consider again Figure 24 (parts b and c). The structure of the left side in Figure 24c accounts for structural constraints, while the following formula $\mathcal{F}$ accounts for TCs and VBCs:

$\mathcal{F}$ = `$A.tag=A AND $B.tag=B AND $C.tag=C AND $D.tag=D AND $E.tag=E AND $F.tag=F AND $G.tag=G AND $H.tag=H AND $D.val>100 AND $C.val=$F.val`

The *evaluation* of the constructed TPQ generates an ordered sequence of bindings (i.e. n-uples of those instance nodes that satisfy the TPQ). In the sequel, we will not further discuss the order of instance nodes, but we assume it to be the consequence of preserving the order of instance nodes within the original XML document.

4.2 Evaluation of TPQs

This section discusses the use of TPQs for determining the fragment sets that are transported from the source part to the construct part as an effect of binding edges. If there is more than one binding edge, the edges may influence one another; this influence is ruled according to a partial order, which is explained next.

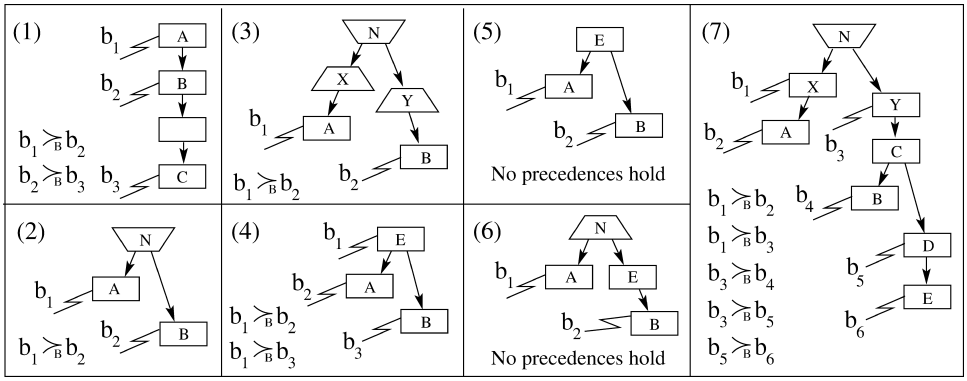


Fig. 25. Order of binding edges: the $\succ_{\mathcal{B}}$ partial order.

4.2.1 *Ordering of Binding Edges.* Binding edges are partially ordered according to the topology of the construct part—according to their right-hand-side adjacent nodes. Such partial order represents the fact that the evaluation of a binding edge b_i is sensitive to an already evaluated edge b_j only if the node $\mathrm{N}_j \in N^C$ bound by b_j is *an ancestor* of the node $\mathrm{N}_i \in N^C$ bound by b_i or N_i and N_j are involved in *the same Cartesian product*. More formally, given two binding edges $b_A, b_B \in \mathcal{B}$ with $A, B \in N^C$, we define the $\succ_{\mathcal{B}}$ partial order as a *transitive relation* such that $b_A \succ_{\mathcal{B}} b_B$ if any of the following holds:

(1) B is a descendant of A with respect to the containment relationship;
(2) A and B are descendants of the same single-tag T-Node N, the path from N to A is at the left of the path from N to B, and the nodes on such paths are all set-tag T-Nodes.[10]

Figure 25 shows some examples of XQBE construct graphs; each case also lists all the (direct) precedences in $\succ_{\mathcal{B}}$. Cases (1–3) apply the above definition, cases (4–6) show that $\succ_{\mathcal{B}}$ is only a partial order, and case (7) shows a configuration in which precedences depend on the transitivity of $\succ_{\mathcal{B}}$. Quite straightforwardly, $\succ_{\mathcal{B}}$ can be also regarded as a partial order on the nodes adjacent to the binding edges, either in the source or in the construct part.

4.2.2 *Computation of TPQs.* An evaluation step of the TPQ occurs whenever the algorithm that explores the construct part requires the evaluation of a binding edge. The result of such evaluation is the set of **distinct** instance nodes associated to the corresponding query node of the source part (more precisely, the distinct images of the variable $V corresponding to the binding edge, which are denoted by $m(\$V)$).

When a binding edge b with k predecessors in $\succ_{\mathcal{B}}$ is under evaluation, those k edges have already been evaluated, and this influences the current

[10]This condition represents the fact that A and B are involved in the same Cartesian product, i.e. that they are the first non-trapezoidal nodes encountered descending two different paths branching out of N (according to the constraints stated in Section 2.1.5).

evaluation,[11] because the corresponding adjacent query nodes are bound to actual values, fixing an *evaluation environment*. The set of bound nodes is modeled by the *env*() function, that returns the full list of the already bound query nodes together with their values. The list is composed of h ≥ k pairs $\langle \$V_i, \overline{n} \rangle$, where $\$V_i \in \mathcal{V}$ and $\overline{n} \in m(\$V_i)$ is the particular instance node associated to the variable according to one binding of the current matching m. Each variable in $\mathcal{V}$ occurs at most once in the list. Note that *env*() may return more than k couples, because fixing the identity of a node may implicitly fix the identity of all its container nodes in the document hierarchy.

The evaluation of a TPQ is denoted by the function call *compute*(N), with $N \in N^S$; this function can be called many times for the same node, within different environments: different assignments of node identities to the already bound query nodes, as returned by the *env*() function.

4.2.3 *Use of Environments for Accessing Ancestor Nodes.* It may happen that compute() is called upon a node that has already been bound. Such situation occurs if $\succ_{\mathcal{B}}$ is opposite to the node containment hierarchy of the source part.

Consider again Figure 24, and assume that nodes X and Y are switched so that the two binding edges cross one another; note that, although unusual, such query is legal and leads to the extraction of all instance nodes $h \in m(\$H)$, which are obtained by the first call to compute(H) and mapped to the new nodes Y. Then, on the second call (compute(G)), G has already been bound. In this situation, the compute call is legal but it is not followed by any computation; instead, the $\langle \$G, \overline{g} \rangle$ couple is retrieved from the list returned by env(), where $\overline{g}$ is the container of the particular $\overline{h}$ being considered at that time.

Thus, environments enable us to recover values that were produced during a previous computation and must be found by ascending the containment relationships in some document that matches the source part.

4.3 Semantics of the Construct Part

The semantics of the construct part is based upon the use of two functions: *constr()*, which recursively visits the construct part and builds the tags of the result, and *eval()*, which visits node instances of the source document and builds the node instances of the result. We first give the general structure of constr(), then define eval(), and then show specific cases of constr() relative to the different node types.

4.3.1 *Types of Node Construction.* We denote the construction of the result by means of a recursive constr() function that accepts as arguments the query node N, which is under evaluation in the recursive visit of the construct part, and an instance node p corresponding to a query node hierarchically above that query node; such instance node is the last one being computed in the recursive depth-first descent of the construct tree.

[11]It will be later clarified that the preceding edges are guaranteed to have already been evaluated at that time, due to the traversal strategy for the construct part.

```
constr (QueryNode N, InstanceNode p) : void
    switch (N)
        case N is a E-Node (□) : constrElement(N, p)
        case N is a set-tag T-Node (△) : constrSet(N, p)
        case N is a single-tag T-Node (▽) : constrSingle(N, p )
        case N is a P-Node (○) : constrPNode(N, p)
        case N is a A-Node (●) : constrANode(N, p)
```

Thus, constr() builds nodes differently, depending upon the type of node of the construct part. The evaluation always starts from the root node of the construct part C with the call constr(root(C), $\perp$), where $\perp$ denotes that at the time of the first call the "previous node under evaluation" is undefined.[12] Before we describe each specific constructor, we introduce the context within the result document and the evaluation function.

4.3.2 *Context Within Result Documents.* Consider the containment hierarchy of the result document, and focus only upon pairs of element nodes. Assume that we are constructing the instance nodes n of a given query node N and that M contains N in the node hierarchy; then, we use the function context(n,M) to denote the instance node of M that contains n. More generally, we refer to the context of n as the vector of all node instances obtained by orderly evaluating the function context over the predecessors of N. The context is progressively defined while the resulting construction takes place, starting from the root node and visiting the construct tree.[13] Similar to environments, contexts are assumed to be globally available and as such are not considered within parameters of our semantic functions.

4.3.3 *Evaluation Function.* The evaluation function is not recursive. It takes as input the current query node being constructed and the instance node of query node hierarchically above, the last one being computed in the recursive descent of the tree. The meaning of the eval() function is then either returning a constant, or a projection of a given node, or the bindings produced by the evaluation of the TPQ. Disregarding the first case, we say that in the second case we "continue" descending a document while in the third case we introduce a discontinuity, and "pass" bindings from the source part.

There are three cases of projection: if the query node is an element, the evaluation returns all the node instances of that element; if it is an attribute, it returns the attribute value; if it is a P-Node, it returns its textual content. We use a standard XPath notation to denote such content.

[12]It is worth noting that root(C) can only be either a node with a binding edge or a T-Node, and if it is a T-Node then all its nearest non-T-Node descendants (reachable along all the descending paths) are guaranteed to have a binding edge. Thus, the construction either starts by adding new tags or by binding data items that originate from the source documents.

[13]Note that the semantics of XQBE requires the notion of an environment in the source part and of a context in the construct part; having these two data structures is inevitable because they maintain the references to node instances that are computed within the source graph and built within the construct graph, and these are independent. Binding edges not only bridge query nodes to instance nodes, but also bridge contexts to environments.

Finally, if the query node M has a binding edge connecting it to a node N in the source part, then eval(M) returns the set of node instances that are computed by invoking compute(N). Using the context, we extract the ancestors of M in the construct part, and then we use the binding edge to derive the corresponding ancestors in the source part. Then, the environment function applies to such ancestors, assigning suitable values to all the variables belonging to the antecedent expressions of N. Given that the env() and context() functions are implicitly available, their use is also implicitly assumed whenever the compute() function is called.

The eval() function is then defined as:

```
eval(QueryNode N, ContextNode c) : list of InstanceNode
    switch (N)
        case N is a V-Node representing a constant k : return { "k" }
        case N has no binding edge
            if N is a E-Node : return { c/N }
            if N is a A-Node : return { att-value( c/@N ) ) }
            if N is a P-Node : return { c/text() }
        case N has a binding edge B : return {compute( N ) }
```

4.3.4 *Detailed Node Construction.* Before discussing the various alternatives in node construction, we introduce some general purpose auxiliary functions. For each query node N of the construct part we define the functions:

Succ(N) : list of QueryNode

returning the list of all the (query) nodes reached by the outgoing arcs of N, in left-to-right order. We also define two sublists of Succ(N), one with attributes, and the other one with all other nodes:

Atts(N) : list of QueryNode
NonAtts(N) : list of QueryNode

4.3.5 *Element Nodes (□).* Element nodes are used to build elements of the result document. Their construction is defined as follows:

constrElement(N, p) : void
$\forall n \in$ eval(N, p)
‘<N’ for $A_1 \ldots A_i \in$ Atts(N) { constr(A_1, n) … constr(A_i, n) } ‘>’
for $S_1 \ldots S_j \in$ NonAtts(N) { constr(S_1, n) … constr(S_j, n) }
‘</N>’

Note that the open and close tags are generated for each node instance n of the query node N. Node instances are found either in the context of p (the predecessor node instance) or by solving the TPQ in correspondence with the binding edge on node N. Note also that attributes are displayed in their proper location, within the opening tag, that the construction is recursively invoked on all the

query subnodes, and that the context to be passed to the recursive constructions is n. Finally, note that, in case of "renaming" (i.e the labels of the nodes connected by a binding edge are different), the output label is determined by the query node of the construct part, and therefore renaming is properly performed.

4.3.6 *Attribute Nodes* (•). Attribute nodes (A-Nodes) are used to build attributes of the result document. Their construction is defined as follows:

constrANode(N, p) : void
    '`N =` {eval(N, p)} '

Note that eval() returns an attribute value that can originate from a constant, a projection, or a binding edge, and the construction is orthogonal to this alternative.

4.3.7 *PCDATA Nodes* (○). PCDATA Nodes (P-nodes) are used to build PCDATA content of the result document. Their construction is defined as follows:

constrPNode(N, p) : void
    '{eval(N, p)}'

Note that p/text() as computed by eval() is a PCDATA value or the empty string, depending on the fact that c is an element node with or without PCDATA content. The text value can also be a constant or the result of a binding edge.

4.3.8 *Set-Tag T-Nodes* (△). Set-tag T-nodes are used to enclose within a pair of tags, the node instances of one or more elements reached by the outgoing arcs. Their construction is defined as follows:

constrSet(N, p) : void
    '<N' for $A_1 \ldots A_i \in$ Atts(N) {constr(A_1, p) ... constr(A_i, p)} '>'
        for $S_1 \ldots S_j \in$ NonAtts(N) {constr(S_1, p) ... constr(S_j, p)}
    '</N>'

Note that, different from the case of constrElement(), the context predecessor node p is passed over without modification. The only action is the insertion of one couple of tags, with continuity of context. If the set-tag T-node includes several elements, their node instances are listed in sequence.

4.3.9 *Single-Tag T-Nodes* (▽). We have left as last, the most complex case of construction, since T-Nodes may introduce Cartesian products. If we consider fragments composed of T-Nodes descending from a given single-tag T-Node N, it is useful to denote as $\text{Succ}^{NT}(N)$, the list of the first non-T Nodes that are reached from N by traversing the fragment depth-first.[14]

[14]Note that, due to syntactic limitations introduced in Section 2, such fragments of T-Nodes only include set-tag T-Nodes without bifurcations.

$\text{Succ}^{NT}(N)$: list of non-T-Nodes

Succ^{NT} can be easily programmed as a recursive function, omitted here for brevity. As for Succ(), we distinguish the attributes and the other kinds of nodes (which in this case can only be E-Nodes or P-Nodes), denoting them as $\text{Atts}^{NT}(N)$ and $\text{NonAtts}^{NT}(N)$. According to the previous notation, let $\text{A}_1 \ldots \text{A}_k \in \text{Atts}^{NT}(N)$ be the attribute query nodes descendants of N and $\text{S}_1 \ldots \text{S}_j \in \text{NonAtts}^{NT}(N)$ the other non-T-Node descendants of N. The construction for single-tag T-nodes is then defined as follows:

constrSingle(N, p) : void
  $\forall a_1 \in \text{eval}(\text{A}_1, p), \ldots \forall a_k \in \text{eval}(\text{A}_k, p), \forall s_1 \in \text{eval}(\text{S}_1, p), \ldots \forall s_h \in \text{eval}(\text{S}_h, p)$
   ‘<N’ {constrOne(A_1, a_1) . . . constrOne(A_k, a_k)} ‘>’
      {constrOne(S_1, s_1) . . . constrOne(S_h, s_h)}
   ‘</N>’

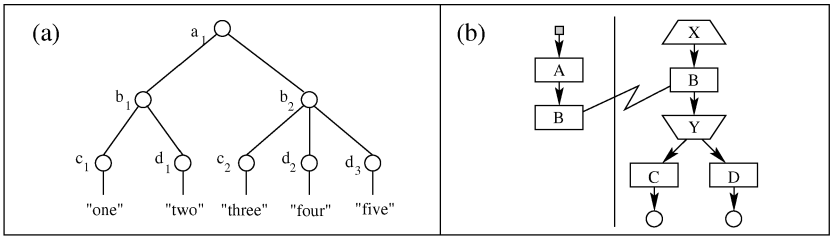
Note that *eval*() must be called in advance, so as to define the sets of instances upon which the Cartesian product is built. These calls produce sets of node instances (when the call concerns attributes or P-Nodes, the corresponding sets are known to be singletons), and every element of each set participates in the Cartesian product with every element of every other set. Note that the recursive call constrOne() is a special constructor, applied to a single node instance rather than a set; this is because the iteration is already performed constructing node N.

ConstrOne() is applied to elements, attributes, P-Nodes, and set-tag T-Nodes without bifurcations (single-tag T-Nodes are excluded as discussed in Section 2):

constrOne(QueryNode N, InstanceNode n) : void
    switch (N)
        case N is a E-Node : ‘<N’ for $\text{A}_i \in \text{Atts}(N)$ { constr(A_i, n) } ‘>’
                                for $\text{S}_j \in \text{NonAtts}(N)$ { constr(S_j, n) }
                            ‘</N>’
        case N is a set-tag T-Node: ‘<N>’ constrOne(succ(N,n)) ‘</N>’
        case N is a P-Node : ‘ { n } ’
        case N is a A-Node : ‘N = { att-value(n) } ’

Note that constrOne() recursively calls itself only in the case of set-tag T-Nodes; otherwise it calls constr() after making one construction step for one specific node instance. Also note that in the recursive call, succ(N) denotes the unique successor to N, according to the syntactic limitation expressed in Section 2.1.5. In all cases, the context of n is passed to the successor constructor in the recursion.

4.3.10 *Example of Construction.* The following example illustrates the semantics of the construct part for a single query. Assume an XML source data represented in Figure 26a and an XQBE query in Figure 26b. We next show the sequence of calls to the construction and evaluation functions, and the result:



constrSet(X, ⊥)	—	<X>
constrElement(B, ⊥)	b1 ∈ eval(B, ⊥)	<B>
constrSingle(Y, b1)	—	<Y>
constrOne(C, c1)	c1 ∈ eval(C, b1)	<C> one </C>
constrPNode("one", c1)	"one" ∈ eval("one", c1)	<D> two </D>
constrOne(D, d1)	d1 ∈ eval(D, b1)	</Y>
constrPNode("two", d1)	"two" ∈ eval("two", d1)	</B>
constrElement(B, ⊥)	b2 ∈ eval(B, ⊥)	<B>
constrSingle(Y, b2)	—	<Y>
constrOne(C, c2)	c2 ∈ eval(C, b2)	<C> three </C>
constrPNode("three", c2)	"three" ∈ eval("three", c2)	<D> four </D>
constrOne(D, d2)	d2 ∈ eval(D, b2)	</Y>
constrPNode("four", d2)	"four" ∈ eval("four", d2)	<Y>
constrSingle(Y, b2)	—	<C> three </C>
constrOne(C, c2)	c2 ∈ eval(C, b2)	<D> five </D>
constrPNode("three", c2)	"three" ∈ eval("three", c2)	</Y>
constrOne(D, d3)	d3 ∈ eval(D, b2)	</B>
constrPNode("five", d3)	"five" ∈ eval("five", d3)	</X>

Fig. 26. Example of construction.

5. A TRANSLATION AND EXECUTION ENVIRONMENT FOR XQBE

In this section, we discuss the implementation of the visual interface of XQBE and the compilation and execution of XQBE queries. Our system translates XQBE into XQuery and then uses existing XQuery engines for running the query. The translation addresses the full XQBE language, and is consistent with the semantics of CoreXQBE given in Section 4. Given that the translator applies a systematic transformation, the resulting XQuery expressions take a general format, called *canonical XQuery form*.

5.1 Translation of XQBE into XQuery

The subset of XQuery adopted for the translation of XQBE queries is defined by means of an EBNF grammar, shown in Figure 27; terminals are enclosed in single apexes and nonterminals are enclosed in angle braces. While we describe the grammar's production, we also describe the canonical XQuery form.

A query (<query>) always translates to *one* XQuery FLWOR expression, constant value, or empty element, possibly contained into an arbitrary number of element constructors ([1]); these constructors would translate a corresponding sequence of set-tag T-Nodes (those with the short edge above) when such a configuration is the root of a tree in the construct part.

```
 [1] <query>         ::= <flwor_expr> | <startTag>‘{’<query>‘}’<endTag> | <emptyTag> | <const>
 [2] <flwor_expr>    ::= <for_clause> <where_clause>? <order_clause>? <return_clause>
 [3] <for_clause>    ::= ‘for’ <var_binding> ( ‘,’ <var_binding> )*
 [4] <var_binding>   ::= ‘$’<var_name> ‘in’ <path_expr>
 [5] <where_clause>  ::= ‘where’ <ex_quantifier>? ‘(’ <conjunction> ‘)’
 [6] <ex_quantifier>::= ‘some’ <var_binding> ( ‘,’ <var_binding> )* ‘satisfies’
 [7] <conjunction>   ::= ( <atom_list> | <neg_clause> ) (‘and’ <neg_clause>)*
 [8] <atom_list>     ::= <atom> ( ‘and’ <atom> )*
 [9] <atom>          ::= <pred_term>   |   ‘exists(’ <path_expr> ‘)’
[10] <pred_term>     ::= <expression> <comparator> <expression>
[11] <expression>    ::= <const> | <variable> | <computation> | <aggregate>
[12] <comparator>    ::= ‘=’  |  ‘<’  |  ‘>’  |  ‘<=’  | ‘>=’  |  ‘!=’
[13] <neg_clause>    ::= ‘not(’ <ex_quantifier>? ‘(’ <atom_list> ‘))’
[14] <return_clause>::= ‘return’ ( <emptyTag> | <path_expr> | <computation> | <aggregate> ) |
                        ‘return’ <startTag> <project_list> <endTag>
[15] <project_list> ::= <startTag> <project_list> <endTag> <project_list> |
                        <emptyTag> <project_list> | ‘{’ <path_expr> ‘}’ <project_list> |
                        ‘{’ <flwor_expr> ‘}’ <project_list> |
[16] <order_clause> ::= ‘order by (’ <name> ( ‘,’ <name> )* ‘)’ (‘ascending’ | ‘descending’)?
[17] <startTag>     ::= ‘<’  <name> <attrList>? ‘>’
[18] <endTag>       ::= ‘</’ <name>  ‘>’
[19] <emptyTag>     ::= ‘<’  <name> <attrList>? ‘/>’
[20] <attrList>     ::= ( <name> ‘="’ ( <path_expr> | <const> | <flwor_expr> ) ‘"’) +
[21] <name>         ::= any valid name for XML elements
[22] <path_expr>    ::= XPath expression (without filtering steps)
[23] <variable>     ::= any variable that has been already bound in an outer <var_binding>
[24] <const>        ::= a text constant to be interpreted as a string or a numeric value
[25] <computation>  ::= an arbitrary arithmetic expression of <variable>s and <const>s
[26] <aggregate>    ::= the invocation of an aggregate function
[27] <var_name>     ::= a valid, automatically generated, unique name for a variable
```

Fig. 27. EBNF specification of the XQuery normal form for XQBE.

The *for* clause of a FLWOR expression ([2]) contains a list of variable bindings ([3]). Variables are bound to a restricted class of path expressions (<path_expr>, [21]), that do not allow filtering steps, because we chose to collect all the predicative conditions in **where** clauses.

The *where* clause has a quite fixed structure ([5–13]): the positive part of the clause is in a conjunctive “prenex” normal form (with all the quantifiers first) and can have as conjuncts, several negated clauses, which in turn are in the same normal form (without any further nested negation). The atomic terms within these clauses are comparisons of values, aggregates or arithmetic expressions ([9–12]). The choice of this particular form is due to the translation algorithm: a **where** clause is typically built by composing several subclauses, each corresponding to a part of the source graph; the prenex conjunctive form helps the composition of such clauses by collecting the variables and the conjunctive atoms directly form the query graph (such components are prepared in the preprocessing phase).

The *return* clause can generate a computed value or a new (possibly empty) element ([14]), which contains a “project list” ([15]). This name recalls that the return clause can project “in breadth and depth” the elements extracted by the **for** clause; it consists of a list of path expressions and nested FLWOR expressions.[15]

[15]Note that attributes are either built by means of $< variable >/@< name >$ path expressions, in case of projection ([15]), or explicitly constructed within the tags ([17, 19]), in case of constants or value-passing binding edges.

A semantic constraint applies to the productions [1], [14], and [15]: the opening and closing tags must match, have the *same* name, so as to guarantee the well-formedness of the generated XML data.

Note that, for the sake of readability, we did not use the XQuery generated by our grammar of Figure 27 in the examples of Section 3, but rather we used the style of W3C Use Cases. The translation algorithm, when applied to the corresponding XQBE queries, generates their Canonical form, which is equivalent but different (an example is given next).

The XQuery translation is built by means of nested FLWOR expressions. The nesting structure of these expressions mimics the calls to the constr() function as defined in 4.3, mapped on the structure of the construct part of the query. Such FLWOR expressions derive from a construction equivalent to that described in Sections 4.1 and 4.2. Intuitively, the constr() functions correspond to XQuery FLWOR expressions, where each "for each" operator in the semantics maps to a **`for`** clause and the constructed couples of tags correspond to node constructors in the **`return`** clause. The recursive calls to constr() functions, which occur within the tags, correspond to nested XQuery expressions.

The native interpretation of XPath expressions within XQuery and the evaluation of the FLWOR expressions themselves correspond to the calls to the eval() function. Nesting of expressions within the `return` clause is done differently, depending on the nature of the nodes in the construct part that correspond to the expressions.

To exemplify the canonical XQuery form and informally discuss the translation process, we show the automatically generated translation of query q14 from Section 2.2.8 (it can also be seen in Figure 29):

```
<books-with-prices> {
 for $entry in doc("www.amazon.com")//entry
 where ( some $book in doc("www.bn.com")//book satisfies
         some $price in $book/price satisfies
         some $price_3 in $entry/price satisfies
         some $title_2 in $book/title satisfies
         some $title in $entry/title satisfies
            $title = $title_2 )
 return <book-with-prices> {
            for $title_4 in $entry/title
            return <title> { $title_4/text() } </title> ,
            for $price_3 in $entry/price
            return <price-amazom> { $price_3/text() } </price-amazom> ,
            for $book in doc("www.bn.com")//book,
                $price in $book/price
            where ( some $title_2 in $book/title satisfies
                    some $title in $entry/title satisfies
                        $title = $title_2 )
            return <price-bn> { $price/text() } </price-bn>
      } </book-with-prices>
} </books-with-prices>
```

The set-tag T-Node in the construct part (`<books-with-prices>`) is translated into a node constructor, whose content derives from the evaluation of the

topmost binding edge and from the "projection and enrichment" of such result. This evaluation is expressed by means of a FLWOR expression, which translates the evaluation of the TPQ in the source part; more precisely, the TPQ is solved to retrieve m($entry), and the full formula $\mathcal{F}$ is translated into the *where* clause of the FLWOR expression, which imposes the existence of the two prices and the join condition on titles.

In the *return* clause each extracted `entry` is renamed to `<book-with-prices>` and projected to its title and price (in turn projected to their PCDATA content). Both these projections map to nested FLWOR expressions. A third nested FLWOR expression translates the last evaluation of the TPQ for extracting the instance nodes corresponding to the `price` query node on the `bn` side. At the time of this last evaluation, the value for the `$entry` variable is taken from env(), as it is already fixed. This corresponds to the fact that the variable `$entry` is used to define the variable `$book` without being redefined, as the current FLWOR expression is inside the scope of the outmost FLWOR expression.

5.2 Implementation of XQBE

XQBE is fully implemented in a tool environment published on the Web [Braga and Campi 2003b]. The implementation is based on a client-server architecture and consists of about 120 Java classes. Along with the core capabilities of drawing XQBE queries and generating their translation into XQuery, we have also included a considerable infrastructure both for supporting the user in the editing process and for program tracing and debugging. The client-server architecture allows us to maintain the translation algorithm without annoying users with patches. In a final version it will be removed.

5.2.1 *Architecture of the XQBE System.* In our client-server implementation, the client provides a visual editor for the user, the server operates the translation and then, if requested, executes the query by invoking an XQuery engine. The client and the server communicate by exchanging an internal representation of queries in an intermediate format that is basically an XML description of the XQBE graphs. An overall picture of the XQBE system architecture is in Figure 28.

The *client* is mainly devoted to the visual editing of the queries. The user is assisted with a strong syntactic feedback, that prevents the composition of incorrect queries. A schema-driven editing mode is also available, which allows the user to compose the XQBE graphs with few mouse clicks, guided by available XML schema or DTD specifications of the target documents. These features are discussed in more detail in Section 5.2.2.

When a query is completed and sent to the server to be processed, the application protocol allows one to request either a simple translation, so that the server returns an XQuery statement, or the query execution as well, so that the server also returns the resulting XML data. If the client only asks for the translation, the returned XQuery statement can be executed on local XQuery engines; this mode is typical for queries targeted to local or private data, while the remote execution is typical for queries upon data available on the

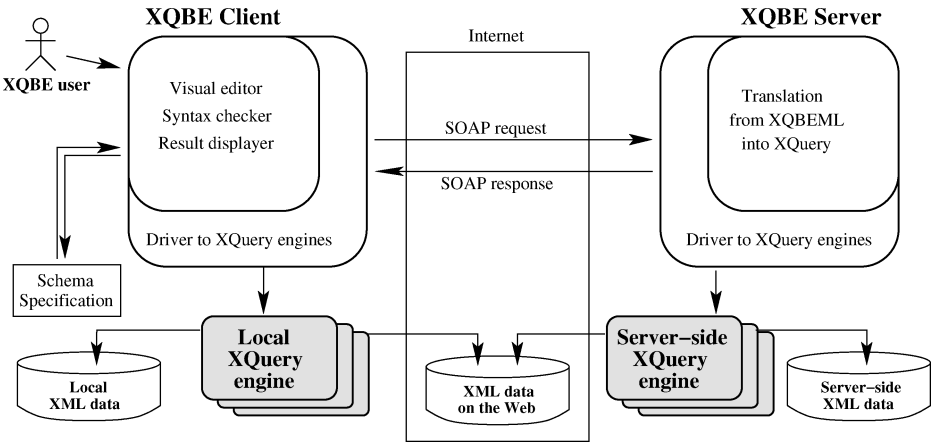


Fig. 28. Overall architecture of the XQBE system.

Web. The XQBE client packs the query represented in XML into a SOAP message with suitable parameters to characterize the request, and sends it to the server.

The *server* is implemented as a Web service capable of translating the XML specification of an XQBE query into an XQuery statement. When the server receives the SOAP request, it unpacks the query, executes the translation algorithm, and then sends a SOAP response containing the generated XQuery statement. The part of the server that executes the translation is the core of our architecture; the translation principle mimics the semantics of XQBE according to the mapping described in Section 5.1.

The server is also capable of executing the generated statement on a server-side XQuery engine. This optional feature is controlled by one of the parameters of the SOAP message. The query can be executed by invoking the APIs offered by several different third party query engines. The XML data produced as the result of the execution is packed in the SOAP response and sent back to the XQBE client, in addition to the XQuery statement, which is always included in the response.

5.2.2 *The Visual Interface.* This section briefly describes the features of our implementation of the XQBE visual interface (a snapshot showing q14 from Section 2.2.8 is in Figure 29).

The XQBE client is a Java stand-alone application that provides an editing interface similar to that of many editors for visual languages based on graphs. Users draw queries in windows composed of two parts, corresponding to the source and construct parts. Graphs are built by choosing the graphical constructs from the toolbar on the left—any portion of these graphs can be cut and pasted from a query to another—and the queries can be compiled and executed with a single click.

The tool also offers the possibility of associating textual annotations to the queries. This feature mainly aims at helping unskilled users, but also tries to make XQBE queries more self-explanatory, capturing the semantics and

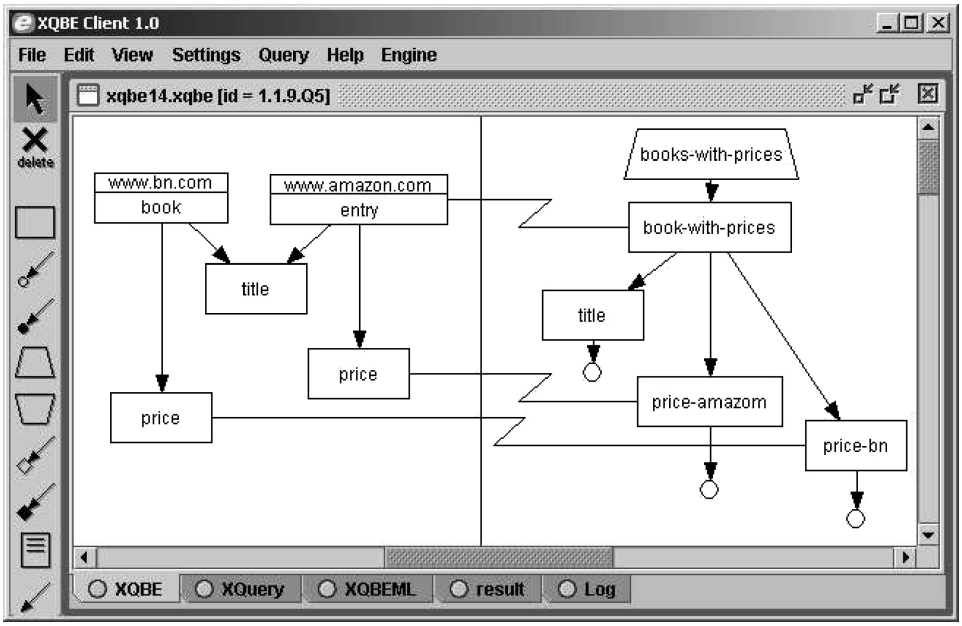


Fig. 29. A snapshot of our interface.

the motivation of a query and helping the exploitation of the environment for educational purposes. In the examples distributed with the downloadable tool, the annotations contain the natural language specification of the queries and a short explanation of the constructs used.

The tool assists the user in many ways during the editing process, and provides syntactic feedback in several forms, to facilitate the building of correct queries. Most incorrect configurations are prevented by the tool while editing; other feedbacks are provided at query compilation time. The syntactic feedback is not limited to detecting "topological" errors, but provides default automatic and semi-automatic corrections to typical or frequent errors, both during the query editing process and at compile time. For some "typical" errors, the tool operates default automatic corrections, however always warning the users of its intervention, so that they might discard the proposed modifications.

The tool allows the user to build the graphs of a query by using a guided construction: users can load *DTD* or *XML Schema* definitions for the target data, thus enabling the tool to suggest the allowed subelements of each selected item, with a one-level expansion. Users can "confirm" the suggested components with a single click, and recursively expand them in turn, thus incrementally constructing the query trees with a "navigational" approach. The suggestions take into account the cardinality and mutual exclusion constraints. Sequences of repeated items are iteratively inserted by clicking on special "element generators," represented by icons like the `author` one in Figure 30, so as to emphasize their multiplicity. The tool also prevents the user from confirming a subelement if it is in mutual exclusion with another already confirmed subelement; if such

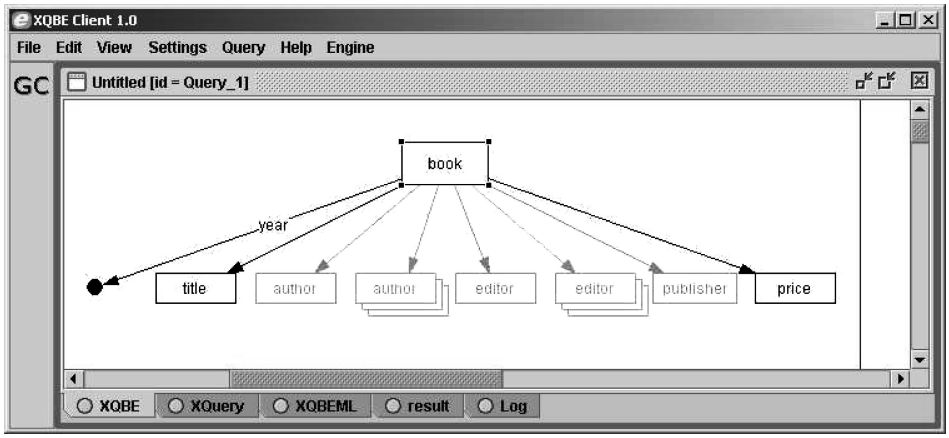


Fig. 30. Schema-guided composition.

subelement is later discarded from the query, the forbidden subelement becomes "confirmable" again.

6. CONCLUSIONS

In this article, we presented XQBE, a visual query language for expressing a large subset of XQuery. We showed how a user can express queries over sets of documents by exploiting the well-understood mapping of XML structures to trees. We also presented a prototype environment for XQBE offering a user-friendly interface that supports the translation of XQBE into XQuery, which can be used in conjunction with any XQuery engine. We trust that our contribution may stimulate academic and industrial research in the field of visual XML languages and interfaces, that we deem fundamental to the spreading of XML to a wider audience.

For our future work, we envision many opportunities of specialization of the language with constructs, primitives, and capabilities that are specific to particular applicative domains. One such specialization has already been developed in the context of fast and efficient access to digital libraries with semi-structured data—more information is available at Braga and Campi [2003a]. While we seek other similar opportunities for spreading the use of XQBE outside of our group, we will also further develop the tool environment, connect it to more XML repository systems, and add query design tools and wizards so as to further improve the ease of use of XQBE.

APPENDIX

A. COMPARISON BETWEEN XQBE AND XML-GL

XQBE can be considered as an evolution of XML-GL [Comai et al. 2001]. XML-GL is a stand-alone graphical query language, while XQBE is specifically targeted to be a suitable visual interface for XQuery. The two languages share the principle of using a source and a construct graph, connected by edges. Given the special focus on XQuery, many constructs of XML-GL have been revised, and

several new concepts have been introduced. In the sequel, we describe some of the differences.

Some aspects of XML-GL have been made more explicit in XQBE. Binding edges have been made explicit instead of relying on the correspondence of node labels, so that the relationships between the involved elements are always under direct control of the user. The construction of newly generated items is expressed by means of new constructs, thus distinguishing between the projection of the extracted items and the insertion of new components; XML-GL instead relied on the availability of a schema specification or the analysis of the whole instance to guess whether an element was "invented" or a projection of another element.

Some other XQuery-specific constructs have required extending XML-GL. For instance, XML-GL did not allow sorting the generated documents, while XQBE includes S-Nodes for this purpose. Similarly, ghost nodes and conditional nodes have been introduced because XML-GL does not provide the capability of specifying conditions in the construct part, nor projecting "in depth" the extracted fragments.

Finally, some elements of XML-GL have been revised; for instance, "reference" arcs have been suppressed and replaced by explicit joins between attributes (as in q16). The reason for this is that the XML-GL notation didn't allow specifying the name of the referenced attribute (that of type ID) and XQuery does not provide a simple way to identify the target of a reference without that knowledge.

While all the above features concern the language design, another noticeable difference between XQBE and XML-GL concerns the semantics; while the style of XML-GL semantics is based on graph matching and on an implicit binding propagation mechanism, reminiscent of logic programming, we opted for an operational semantics, which explains how XQBE results can be "computed" by inspecting the target documents and also how XQBE expressions can be translated into XQuery.

In many queries typically supported by XQuery, XQBE is much simpler than XML-GL. Queries like q4, q5, and q6 could hardly be expressed in XML-GL, and in any case would be very cryptic, due to the limitation of the projection features of XML-GL. For example, assume a different structure of the `book` elements in the running example, with a (possibly empty) sequence of reviewers' remarks:

```
<!ELEMENT book (title, (author+ | editor+), reviewer*)>
<!ELEMENT reviewer ( last, first?, remark+ )>
```

Then the following query cannot be easily and compactly expressed in XML-GL:

```
for $b in //book
return  <book>
          { $b/editor/affiliation }
          { $b/author/last }
          { $b/reviewer/remark }
        </book>
```

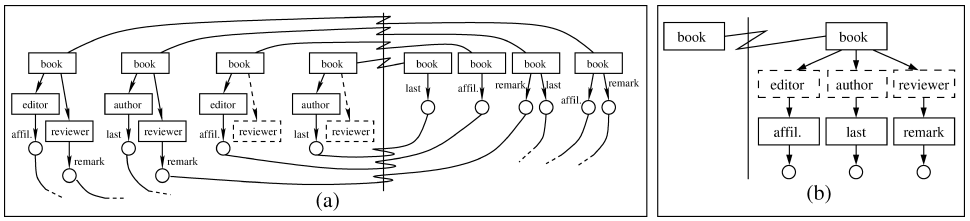


Fig. 31. Comparison between XML-GL and XQBE.

The tricky point is that all `book` elements should contain the (possibly empty) sequence of editor affiliations, the (possibly empty) sequence of authors' surnames and the (possibly empty) sequence of reviewers' remarks. Thus, all the different cases must be distinguished in XML-GL, taking into account the (optional) presence of some reviewers and the (mutually exclusive) presence of authors or editors. The XML-GL formulation of this query is shown in Figure 31a. Instead, XQBE allows for a much simpler and more compact representation (Figure 31b), because it allows us to factorize the constraints in a breadth/depth projection.

ACKNOWLEDGMENTS

We wish to thank Sara Comai and Letizia Tanca for their interesting discussions and useful suggestions, while Enrico Augurusa, Alessandro Raffio, Luca Lulani and Massimo Sarchi deserve our gratitude for contributing to the implementation. We are also very grateful to the anonymous reviewers for their excellent feedback and suggestions.

REFERENCES

AMER-YAHIA, S., CHO, S., LAKSHMANAN, L. V. S., AND SRIVASTAVA, D. 2001. Minimization of tree pattern queries. In *SIGMOD Conference*.

ANGELACCIO, M., CATARCI, T., AND SANTUCCI, G. 1990. Qbd*: A graphical query language with recursion. *IEEE Trans. Soft. Eng. 16*, 10, 1150–1163.

ATZENI, P., MECCA, G., AND MERIALDO, P. 1997. To weave the web. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*.

BONGIO, A., CERI, S., FRATERNALI, P., AND MAURINO, A. 2001. Modeling data entry and operations in WebML. *Lecture Notes in Computer Science 1997*.

BOUGANIM, L., CHAN-SINE-YING, T., DANG-NGOC, T.-T., DARROUX, J. L., GARDARIN, G., AND SHA, F. 1999. Miro web: Integrating multiple data sources through semistructured data types. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99), Edinburgh, Scotland, UK*. 750–753.

BRAGA, D. AND CAMPI, A. 2003a. BiblioXQBE. `http://dbgroup.elet.polimi.it/biblioXQBE`.

BRAGA, D. AND CAMPI, A. 2003b. XQBE Web Site. `http://dbgroup.elet.polimi.it/XQBE`.

CAREY, M., HAAS, L., MAGANTY, V., AND WILLIAMS, J. 1996. Pesto: An integrated query/browser for object databases. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB'96)*, D. McLeod, R. Sacks-Davis, and H. Schek, Eds. 203–214.

CHAWATHE, S., BABY, T., AND YEO, J. 2001. Vqbd: Exploring semistructured data (demonstration description). In *Proceedings of the ACM SIGMOD*. 603.

COHEN, S., KANZA, Y., KOGAN, Y. A., NUTT, W., SAGIV, Y., AND SEREBRENIK, A. 1999. Equix easy querying in XML databases. In *WebDB (Informal Proceedings)*. 43–48.

COHEN, S., KANZA, Y., KOGAN, Y. A., NUTT, W., SAGIV, Y., AND SEREBRENIK, A. 2000. Combining the power of searching and querying. In *5th International Conference on Cooperative Information Systems*.

COMAI, S., DAMIANI, E., AND FRATERNALI, P. 2001. Computing graphical queries over xml data. *ACM TOIS 19*, 4, 371–430.

COMAI, S., DAMIANI, E., POSENATO, R., AND TANCA, L. 1998. A schema based approach to modeling and querying www data. In *FQAS'98*. 110–125.

CONSENS, M. P. AND MENDELZON, A. O. 1990. The g+/graphlog visual query system. In *Proceedings of the 1990 ACM SIGMOD, Atlantic City, NJ, May 23–25*. 388.

CRUZ, I. F., MENDELZON, A. O., AND WOOD, P. T. 1987. A graphical query language supporting recursion. In *Proceedings of the ACM SIGMOD*. 323–330.

CRUZ, I. F., MENDELZON, A. O., AND WOOD, P. T. 1988. G+: Recursive queries without recursion. In *2nd International Conference on Expert Database Systems*. 355–368.

FERNANDEZ, M., SIMÉON, J., WADLER, P., CLUET, S., DEUTSCH, A., FLORESCU, D., LEVY, A., MAIER, D., MCHUGH, J., ROBIE, J., SUCIU, D., AND WIDOM, J. 1999a. Xml query languages: Experiences and exemplars. `http://www-db.research.belllabs.com/user/simeon/xquery.ps`.

FERNANDEZ, M., SUCIU, D., AND TATARINOV, I. 1999b. Declarative specification of data-intensive web sites. In *Proceedings of the Workshop on Domain Specific Languages (DSL)*.

FILHA, I. M. R. E., LAENDER, A. H. F., AND DA SILVA, A. S. 2001. Querying semistructured data by example: The qsbye interface. In *Workshop on Information Integration on the Web*. 156–163.

IVES, Z. G. AND LU, Y. 2000. Xml query languages in practice: an evaluation. In *Proceedings of WAIM'00*. 29–40.

JAESCHKE, G. AND SCHEK, H. J. 1982. Remarks on the algebra on non first normal form relations. In *Proceedings of 1st ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*. 124–138.

KEPSER, S. 2002. A proof of the Turing-completeness of xslt and xquery. Technical report SFB 441, Eberhard Karls Universitat Tubingen. May.

LAKSHMANAN, L. V. S., RAMESH, G., WANG, H., AND ZHAO, Z. J. 2004. On testing satisfiability of tree pattern queries. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*.

LEVY, A. Y., RAJARAMAN, A., AND ORDILLE, J. J. 2002. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB)*.

LUDAESCHER, B., PAPAKONSTANTINOU, Y., VELIKHOV, P., AND VIANU, V. 1999. View definition and dtd inference for xml. In *Proceedings of the Post-IDCT Workshop*.

MUNROE, K. AND PAPAKONSTANTINOU, Y. 2000. Bbq: A visual interface for browsing and querying xml. In *Proceedings of the 5th Working Conference on Visual Database Systems*. 277–296.

PAPAKONSTANTINOU, Y., PETROPOULOS, M., AND VASSALOS, V. 2002. Qursed: Querying and reporting semistructured data. In *Proceedings of the ACM SIGMOD*.

PAREDAENS, J., DEN BUSSCHE, J. V., ANDRIES, M., GEMIS, M., GYSSENS, M., THYSSENS, I., GUCHT, D. V., SARATHY, V., AND SAXTON, L. V. 1992. An overview of good. *SIGMOD Record 21*, 1, 25–31.

PAREDAENS, J., PEELMAN, P., AND TANCA, L. 1995. G-log a declarative graph-based language. *IEEE Trans. Knowl. Data Eng.*

PETROPOULOS, M., VASSALOS, V., AND PAPAKONSTANTINOU, Y. 2001. Xml query forms (xqforms): Declarative specification of xml query interfaces. In *Proceedings of the 10th WWW Conference*.

VASSALOS, V. AND PAPAKONSTANTINOU, Y. 2000. Expressive capabilities description languages and query rewriting algorithms. *J. Logic Prog. 43*, 1, 75–122.

W3C. 2001. Extensible Stylesheet Language (XSL). `http://www.w3c.org/TR/xsl/`.

W3C. 2003a. XML Query Use Cases. `http://www.w3.org/TR/xmlquery-use-cases`.

W3C. 2003b. XQuery: An XML Query Language. `http://www.w3.org/XML/Query`.

ZLOOF, M. M. 1977. Query-by-example: A data base language. *IBM Syst. J. 16*, 4, 324–343.

Received August 2003; revised May and October 2004; accepted November 2004