XQBE: A Graphical Interface for XQuery Engines

Daniele Braga
braga@elet.polimi.it

Alessandro Campi
campi@elet.polimi.it

Stefano Ceri
ceri@elet.polimi.it

Dipartimento di Elettronica e Informazione, Politecnico di Milano
Piazza Leonardo da Vinci 32, 20133 Milano, Italy

Abstract

XQuery is increasingly popular among computer scientists with a SQL background, since queries in XQuery and SQL require comparable skills to be formulated. However, the number of these experts is limited, and the availability of easier XQuery "dialects" could be extremely valuable. Something similar happened with QBE, initially proposed as an alternative to SQL, that has then become popular as the user-friendly query language supported by MS Access. We designed and implemented XQBE, a visual *dialect of XQuery that uses hierarchical structures, coherent with the hierarchical nature of XML, to denote the input and output documents.*

In this demonstration we brie¤y introduce XQBE; we then describe how our implementation generates the XQuery translation of the visual queries and also how the XQBE translation of an XQuery statement is generated, provided that XQBE is expressive enough for that query.

1 Introduction

The diffusion of XML sets a pressing need for providing the capability to query XML data to a wide spectrum of users, typically lacking in computer programming skills. This demonstration presents a user friendly interface, based on an intuitive visual query language (XQBE, *XQ*uery *B*y *E*xample), that we developed for this purpose, inspired by the QBE [3]. QBE showed that a visual interface to a query language is effective in supporting the intuitive formulation of queries when the basic graphical constructs are close to the visual abstraction of the underlying data model. Accordingly, while QBE is a relational query language, based on the representation of tables, XQBE is based on the use of annotated trees, to adhere to the hierarchical nature of XML. XQBE was designed with the objectives of being intuitive and easy to map directly to XQuery. Our interface is capable of generating the visual representation of many XQuery statements that belong to a subset of XQuery, de£ned by our translation algorithm (sketched later).

```
<bib>
 <book year="1994">
  <title> TCP/IP Illustrated </title>
  <author> <last> Stevens </last>
           <first> W. </first> </author>
  <pub> Addison-Wesley </pub>
  <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>
  <pub> Morgan Kaufmann Publishers </pub>
  <price> 39.95 </price>
 </book>
</bib>
```

Figure 1. A sample document (bib.xml)

XQBE allows for arbitrarily deep nesting of XQuery FLWOR expressions, construction of new XML elements, and restructuring of existing documents. However, the expressive power of XQBE is limited in comparison with XQuery, which is Turing-complete. The particular purpose of XQBE makes *usability* one of its critical success factors, and we considered this aspect during the whole design and implementation process. Still from a usability viewpoint, our prototype is a £rst step towards an integrated environment to support both XQuery and XQBE, where users freely alternate between the XQBE and XQuery representations.

Our demo consists of several examples of queries in XQBE and shows how our prototype allows to switch between equivalent representations of the same query.

2 XQuery By Example

XQBE is fully described in [1]. Here we only introduce its basics by means of a query (named Q1) that reads "*List books published by Addison-Wesley after 1991, including*

```
<bib>
{for $b in document("bib.xml")/bib/book
 where $b/pub="Addison-Wesley" and $b/@year>1991
 return <book year="{ $b/@year }">
         { $b/title } </book>       }</bib>
```

Figure 2. The XQuery statement for Q1

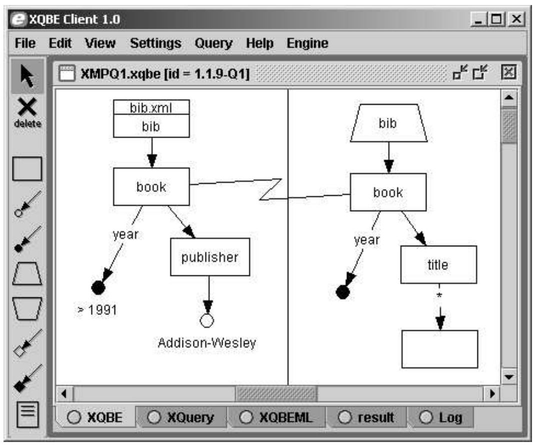

Figure 3. The XQBE version of Q1

their year and title", on the data in Figure 1. Its XQuery version (as proposed by the W3C in [2]) is in Figure 2, while its XQBE version is in Figure 3.

A query in XQBE always has a vertical line that separates the *source* part (the one on the left) from the *construct* part (on the right). Both parts contain labelled graphs that represent XML fragments and express properties of such fragments: the source part describes the XML data to be matched to construct the query result, while the construct part speci£es which parts are to be retained in the result and the new XML items to be inserted. The correspondence between the components of the two parts is expressed by explicit bindings. XML *elements* in the target document are represented as labelled rectangles, their *attributes* are represented as black circles (with the attribute *name* on the arc between the rectangle and the circle), and their PC-DATA content is represented as an empty circle. In Figure 3 the source part matches the `book` elements with a `year` attribute greater than 1991 and a `publisher` subelement equal to "Addison-Wesley". As in the `bib` node, nodes can be labelled with URLs to locate the target XML documents.

In the construct part, the paths that branch out of a bound node indicate which of its contents are to be retained. 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. The trapezoidal `bib` node means that all the generated `book`s are to be contained into one `bib` element.

2.1 Translation from XQBE to XQuery

The translation process takes as input an XQBE query and produces as output its XQuery translation, a sentence of the XQuery subset de£ned by the grammar in £gure 4. The translation starts with a preprocessing of the source part, to compute variable bindings and predicative terms. The query is then generated by processing the construct part with a recursive visit.

Preprocessing

The graphs in the *source part* are parsed to detect those graphical con£gurations, which map to variable de£nitions. These variables are used to construct the predicative terms that express the selection criteria. Each node reached by a *binding edge* causes the instantiation of a variable. This variable is associated to a path expression, which is derived from the path that reaches the node. Variables are also instantiated in correspondence to *bifurcations*. These variables help in enforcing that the items in the branching paths do belong to *the same* ancestor. *Join* nodes (those with *con¤uences*) originate as many variables as their incoming arcs. The predicative terms are generated taking into account the comparator associated to the node. Leaf nodes *with £ltering labels* cause the instantiation of variables to express the selection conditions. Negated branches are visited in the same way, with the only restriction that the visit does not begin until all the positive nodes have been visited.

Processing

A recursive visit of the *construct part* generates a FLWR expression for each node connected by a binding edge. The for clause de£nes the variable instantiated in the node on the other side of the binding. Trapezoidal nodes are translated into node constructors. This recursive visit of the construct part results in an XQuery statement composed of nested FLWR expressions. The where clause of each nested "internal" FLWR expression is assembled with the predicative terms pre-computed in the preprocessing phase. These terms are collected out of the source part, according to criteria dictated by the variables already bound in the for clauses of the "external" FLWR expressions. The generated translation of Q1 is[1]:

```
<bib> {
 for $book in doc("bib.xml")/bib/book
 where (
  some $pub_value in $book/publisher/text() satisfies
  some $year in $book/@year satisfies
    $year > 1991 and $pub_value = "Addison-Wesley" )
 return <book> { $book/@year, $book/title } </book> }
</bib>
```

[1]The generated statement is equivalent to Q1, but in a rather different form, as we use a normalized syntax (that of Figure 4) for the automatically generated translation.

```
 [1] <query>         ::= <flwor_expr> | <startTag>'{'<query>'}'<endTag> | <emptyTag> | <const>
 [2] <flwor_expr>    ::= <for_clause> <where_clause>? <return_clause> <order_clause>?
 [3] <for_clause>    ::= 'for' <var_binding> ( ',' <var_binding> )*
 [4] <var_binding>   ::= <variable> 'in' <path_expr>
 [5] <where_clause>  ::= 'where' <ex_quantifiers>? '(' <conjunction> ')'
 [6] <ex_quantifier> ::= 'some' <var_binding> ( ',' <var_binding> )* 'satisfies'
 [7] <conjunction>   ::= <atom_list>  |  <neg_clause>  |  <atom_list> '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> | <variable> | <computation> | <aggregate> ) |
                         'return' <startTag> <projection_list> <endTag>
[15] <project_list>  ::= <startTag> <project_list> <endTag> <project_list> |
                         '{' <path_expr> '}' <project_list> | '{' <flwor_expr> '}' <project_list> |
[16] <order_clause>  ::= 'order by (' <name> ( ',' <name> )* ')'
```

Figure 4. EBNF speci£cation of the XQuery subset expressible with XQBE

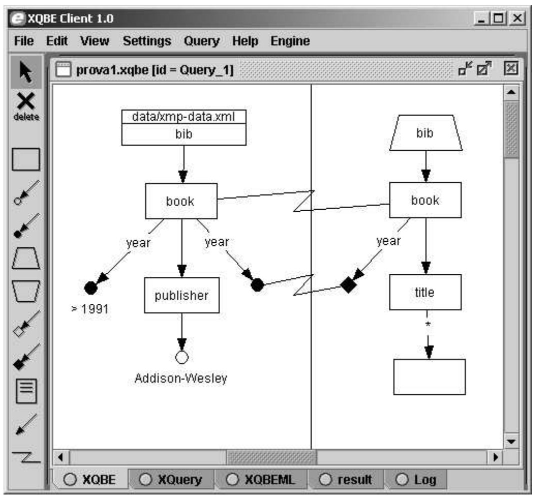

Figure 5. The automatically generated XQBE for Q1

2.2 Translation from XQuery to XQBE

The £rst step to construct the XQBE translation of an XQuery statement is to normalize and parse it. Typical algebraic transformations are applied in this phase, such as, for instance, the conversion of the universal quanti£cation into the existential quanti£cation (the only one supported by XQBE). As an example, consider that the two clauses below are equivalent:

```
where not (every $a in $b/author/first satisfies
          $a/text()= "John")
where some $a in $b/author/first satisfies
     not( $a/text()="John")
```

Once the query is normalized, the XQBE graph is built analyzing its clauses in a precise order. The de£nition of the £rst variable in the for clause allows to construct the £rst chain of nodes in the source part. The other variables and the terms in the where clause allow to construct the remaining branches of the structures in the source part. The return clause contains the information to compute the construct part of the XQBE query and the binding edges. A module in our implementation is dedicated to compute suitable coordinates for the nodes of the generated XQBE, using some heuristics that correspond to "readability" criteria.

3 Conclusions

We have shown a query in XQBE (in Figure 3) equivalent to the W3C query of Figure 2, and described how our interface generates an equivalent XQuery version of that query. We have also shown that the W3C query can be automatically translated into XQBE in a form (in Figure 5).

The contribution of our work is the availability of an environment in which users can query XML data with a GUI, access the generated XQuery statement, and also visualize the XQBE version of a large class of XQuery statements. Moreover they can modify any of the representations and observe the changes in the other representation. Our prototype can thus be considered as a £rst component of an environment for learning XQuery or practising it.

References

[1] D. Braga and A. Campi. A graphical environment to query xml data with xquery. In *Proc. of the 4th WISE*, Roma (Italy), December 2003.

[2] W3C. XML Query Use Cases, August 2003. http://www.w3.org/TR/xmlquery-use-cases.

[3] Moshé M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.