XTree for Declarative XML Querying

Zhuo Chen[1], Tok Wang Ling[1], Mengchi Liu[2], and Gillian Dobbie[3]

[1] School of Computing, National University of Singapore,
Lower Kent Ridge Road, Singapore 119260
{chenzhuo, lingtw}@comp.nus.edu.sg

[2] School of Computer Science, Carleton University,
Ottawa, Ontario, Canada K1S 5B6
mengchi@scs.carleton.ca

[3] Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand
gill@cs.auckland.ac.nz

Abstract. How to query XML documents to extract and restructure the information is an important issue in XML research. Currently, XQuery based on XPath is the most promising standard of W3C. In this paper, we introduce a new set of syntax rules called XTree, which is a generalization of XPath. XTree has a tree structure, and a user can bind multiple variables in one XTree expression. It explicitly identifies list-valued variables, and defines some natural built-in functions to manipulate them. XTree expression can also be used in the result construction part of a query, to make it easy to read and comprehend. With these differences, XTree expressions are much more compact, and more convenient to write and understand than XPath expressions. We also give algorithms to convert queries based on XTree expressions to standard XQuery queries.

1 Introduction

XML is fast emerging as the dominant standard for data representation and exchange in the web. How to query XML documents is an important issue in XML research. There have been various query languages proposed in the past few years, such as XPath[8], XQuery[9], Lorel[1], XML-GL[2], XQL[10], XML-QL[11], XSLT[12], YATL[3], XDuce[4], a declarative XML query language[5], a rule-based query language[6], etc. Some of them are in the tradition of database query languages, others are more closely inspired by XML. The XML Query Working Group has published XML Query Requirements for XML query languages[7], and XQuery has been selected as the basis for an official W3C query language for XML.

Unlike querying on relational databases whose results are always flat relations, the results of queries on XML documents are complex, and need to be formatted explicitly. Thus, XML queries must have two parts: a querying part and a result construction part. The existing XML query languages intermix these two parts in a nested way and thus make queries complicated. For example, XML-QL has two constructs: *where* and *construct*, for querying and result construction respectively. However, the *construct* clause can contain nested *where-construct* clauses so that querying and result-construction are intermixed. Similarly, for XQuery, it uses five constructs: *for*, *let*, *where*, *order by* and *return*, i.e., FLWOR expressions. As in XML-QL, FLWOR expressions can be nested in the *return* clause to form a nested querying structure.

In this paper, we will analyze the limitations of XPath, and propose a new set of syntax rules called XTree, which is a generalization of XPath, and show how it can efficiently replace the notations of XPath. To be compatible with current XQuery parsers, we also give algorithms to convert queries based on XTree expressions to standard XQuery queries.

The rest of this paper is organized as follows. Section 2 introduces the background of XPath and XQuery, and discusses their limitations. Section 3 introduces the XTree syntax with some examples. Section 4 gives algorithms to transform queries based on XTree expressions to standard XQuery queries. Finally, section 5 summarizes this paper and points out future research directions.

2 Background on XPath and XQuery

XPath[8] is a set of syntax rules for defining parts of XML documents. It uses paths to locate nodes (elements and attributes) in XML documents, and the path expressions look very much like computer file system paths. For example, consider the following bibliography XML document in Fig. 1.

```
<bib name=“IT”>
  <book id=“b001” year=“1994”>
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
  </book>
  <book id=“b002” 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</publisher>
  </book>
  <journal id=“j001” year=“1998”>
    <title>XML</title>
    <editor><last>Date</last><first>C.</first></editor>
    <editor><last>Gerbarg</last><first>M.</first></editor>
  </journal>
</bib>
```

Fig. 1. Sample bibliography XML document

Table 1 gives some examples of XPath expressions, according to the XML document in Fig. 1.

Table 1. Sample XPath expressions

XPath expression	Description
/bib/book/@year	get attribute "*year*" of each book.
/bib/book/author	get element "*author*" of each book.
//author	get all elements named "*author*", regardless of their absolute paths.
*/bib/book/**	get all subelements of each book.
*/bib/book/@**	get all attributes of each book.

XQuery[9] is a powerful way to search XML documents for specific information. It is based on XPath and has the FLWOR statements. *For* clause (syntax: "*For $var in xpath-expression*") iterates the variable over the result of the XPath expression, whereas *let* clause (syntax: "*Let $var := xpath-expression*") bind the variable to the result of the XPath expression. XQuery also supports complex queries and complex result constructions with nested clauses.

Although XPath can clearly define a unique path in the XML tree, it has some limitations. Firstly, in an XPath expression, although the condition can be a branch, there is still a linear path to the target node set. Thus, we can only assign one variable for each XPath expression, which is very inefficient. If a query uses several paths, a user must assign a variable to each path.

Example 1. If a user is interested in title, authors and publisher of each book in the bibliography data in Fig.1, we have to write the following statements in querying part:

```
for $b in /bib/book
let $t := $b/title,  $a := $b/author,  $p := $b/publisher
```

Secondly, it is difficult to reveal the relationship among correlated XPaths, as XPath does not show such correlation explicitly. This may cause some mistakes if the user does not pay attention when writing a query.

Example 2. Create a flat list of all the title-author pairs for all the books, and each pair is enclosed in a "result" element. Fig. 2a is a wrong answer, because it does not pay attention to the correlation of the two XPaths. It will produce a Cartesian product of all authors and titles, regardless whether they are of the same book. Fig. 2b gives the correct version of this query. Note that the curly braces *{ }* means enclosed expression in XQuery, which force the inner code to be evaluated and replaced by their value, instead of being treated as literal text.

```
for $t in /bib/book/title,
     $a in /bib/book/author
return
   <result>
      { $t }
      { $a }
   </result>
```

Fig. 2a. Wrong answer

```
for $b in /bib/book,
     $t in $b/title, $a in $b/author
return
   <result>
      { $t }
      { $a }
   </result>
```

Fig. 2b. Correct answer

Thirdly, XPath expressions are only used in the querying part of XQuery, not in the result construction part. For the result construction part of XQuery, XML segments can be written directly in the *return* clauses. The mixture of literal text, enclosed expressions and even nested sub-queries in the result construction make the whole query difficult to read and comprehend.

Finally, XPath can only bind variables on the whole node (attribute or element) structure, which is a name-value pair. If we want to get some substructure (name or value) of the node, we have to invoke some built-in functions (*local-name()* to get node name, *string()* to get string value). Thus it is difficult to query XML documents with unknown structure, or to rename the nodes in the returned result.

3 XTree

We now introduce a new set of syntax rules called XTree, which is a generalization of XPath. It has a tree structure like the structure of XML documents. As in XPath, child node follows parent node via a slash */*, and a double-slash *//* means no matter how many levels down. However, in XTree expressions, sibling tree nodes are enclosed by curly braces *{ }* and are separated by commas, and *{ }* can be nested.

In XTree expressions, we use logical variables as place holders to bind/match the values at their places. Because we need to bind various parts of XML documents to logical variables, in order to be flexible and easy to use in practice, the variables in XTree are *non-typed*. A variable can be used for any location, but when the same variable occurs in different places in the query, it has the same value. There are two kinds of variables: single-valued variables and list-valued variables. *Single-valued variables* start with *$*, such as *$X*. *List-valued variables* are of the form *{$X}* which is constructed from the single-valued variable *$X* that ranges over all *$X* instances in a list. Note that both sibling nodes and list-valued variables are enclosed by curly braces, but it is very easy to differentiate them. Sibling nodes will have commas as separators in the braces, whereas list-valued variables do not have comma in braces.

In XTree expressions, symbol → is used to assign values to variables, it is used in the querying part only; and symbol ← is used to get values from variables, it is used in the result construction part only. Because we can write the result construction part as an XTree expression, we do not need to use curly braces *{ }* to indicate the enclosed expression, or nested query blocks.

3.1 XTree in Querying Part

In the querying part of a query, XTree expressions can be used to bind variables on some specific sets of nodes. Symbol → will assign values of nodes on the left side to the variable on the right side. If the right side is a single-valued variable, it means to iterate the node values one by one, as in *for* clause of XQuery; if the right side is a list-valued variable, it means to keep all node values in the list, as in *let* clause of XQuery.

Example 3. For the XML document in Fig. 1, if we are interested in the *year* and *title* of each book, and its authors' *last* names and *first* names, we can use the variables *$y*, *$t*, *$first*, *$last* to bind them respectively as follows:

bib/book/{@year→$y, title→$t, author/{last→$last, first→$first}}

In this way, we can instantiate many variables in one XTree expression, while each XPath expression can only instantiate one variable. The above XTree expression corresponds to the following 6 XPath expressions in XQuery:

```
for $book in /bib/book,
    $y in $book/@year, $t in $book/title, $author in $book/author,
    $last in $author/last, $first in $author/first
```

Example 4. XTree allows a user to use path abbreviations as in XPath, suppose we want to get the last name and first name elements at whatever depth in the document, we can write the following XTree expression:

/bib//{last→$last, first→$first}

Here the curly braces *{ }* enclosing two elements *last* and *first* specifies that these two elements are siblings, they share a common parent, which is */bib/book/author* or */bib/journal/editor* according to the sample XML document in Fig. 1.

XTree also allows a user to bind variables on the structure of XML document, that is, a user can write variable *$var* on the left side of → symbol, and here *$var* will be bound to the name of the corresponding element or attribute.

Example 5. If we want to obtain some attribute with value "2000" in some book element, and bind variable $b to that book, we can use the following expression:

/bib/book→$b/@$attr="2000"

According to the sample XML document in Fig. 1, *$b* will bind to the second book element, and *$attr* will bind to string "year", which is the attribute name. The XQuery version of this query is as following, which is more complicated (note the use of built-in functions to split the name-value pair of variable *$attr*):

```
for $b in /bib/book,
    $attr in $b/@*
where string($attr) = "2000"
return local-name($attr)
```

3.2 List-valued Variables

List-valued variables are like the variables in *let* clauses of XQuery, however in XQuery these variables look exactly like single-valued variables. In XTree, list-valued variables are explicitly indicated by curly braces *{ }*. Also, unlike XPath and XQuery which use many unintuitive functions, we define some natural functions that are obvious and easy to understand, and they are used in object-oriented fashion.

Example 6. Suppose list-valued variable *{$numbers}* binds to a list of numbers, then we can obtain their aggregate values as follows:

{$numbers}.count()	returns the number of items in the list
{$numbers}.avg()	returns the average value of items in the list
{$numbers}.min()	returns the minimum value in the list
{$numbers}.max()	returns the maximum value in the list
{$numbers}.sum()	returns the sum of values in the list

Example 7. Suppose list-valued variable *{$names}* bind to a list of *name* elements, then we have the following built-in functions and operators:

{$names}[1-3, 6]	returns a sublist containing 1st to 3rd items, and the 6th item
{$names}.last()	returns the last item in the list
{$names}.sort()	sorts the items in the list in ascending order
{$names}.sort_desc()	sorts the items in the list in descending order
{$names}.distinct()	eliminates duplicate items in the list
{$names}.random(3)	picks out 3 items randomly
$name Î {$names}	checks whether an item is in the list
{$names'} Í {$names}	checks whether the first list is a sublist of the second list

Next we will define the semantics of list-valued variables.

Definition 1. The *associated path* of variable *$a* (or *{$a}*) is the absolute path expression from root to the nodes represented by *$a* (or *{$a}*).

For example, in XTree expression */bib/book→$b/title→$t*, the *associated path* of *$t* is */bib/book/title*.

Definition 2. Variable *$a* is an *ancestor variable* of *$b* if *$a* and *$b* are defined in the same XTree expression, and the associated path of *$a* is a prefix of the associated path of *$b*.

For example, in XTree expression */bib/book→$b/{title→$t, author→$a}*, *$b* is an *ancestor variable* of *$t* and *$a*, but *$t* is not an *ancestor variable* of *$a*.

Definition 3. In an XTree expression, when a variable is bound to a value in the query evaluation, the variable is *instantiated.*

For example, in XTree expression */bib/book/{author→$a/first→$first, title→$t}*, in the evaluation, when we have reach */bib/book/author*, *$a* is *instantiated*; when reach */bib/book/author/first*, *$first* is in*stantiated*.

Definition 4. The *value* of list-valued variable *{$a}* is a list of all instances of *$a* with all its ancestor variables instantiated.

Example 8. Compare the following two XTree expressions:

XTree expression	**Value of *{$a}***
/bib/book/author→{$a}	all the author elements of all the books.
/bib/book→$b/author→{$a}	all the authors of a certain book *$b*. When *$b* is bound to next book, *{$a}* will also bind to the authors of next book.

Note that in the first expression, value of *{$a}* is the concatenation of all the authors of each book, which may include duplicated authors (an author may write several books). We can use the function *{$a}.distinct()* to remove duplicates.

3.3 XTree in Result Construction Part

XTree expressions not only can be used to bind variables in the querying part, but also can be used to define the result format. We use symbol ← to get values of variables from right side and assign them to the expressions on the left side. If the right side is a single-valued variable, we just put its value of current iteration to the left side expression; if the right side is a list-valued variable, we will put each value in the list to the left side expression. Unlike the *return* clause in XQuery that often mixes XML plain text, enclosed expressions and even sub-queries, here the result construction part is just an XTree expression without nesting, which is very simple and easy to read.

Note that unlike the XTree expressions in the querying part, which allow conditions and abbreviations (such as // for any levels down, * for all sub-elements, @* for all attributes), the XTree expression in the result construction part must be a concrete one, which does not allow any condition checking or uncertainty in the structure.

The query based on XTree expressions is similar to XQuery, and it has the QWOC (*Query-Where-Order by-Construct*) statements. *Query* clause contains one or more XTree expressions for selection and variables binding; *where* clause and *order-by* clause are optional, they are used for constraints and ordering respectively; *construct* clause contains one XTree expression to define the output format, it does not have a nested structure as the *return* clause in XQuery.

Example 9. If we want to list the titles and publishers (but not authors) of books which are published after 1995, we can write the query as follows:

query /bib/book/{@year→$y, title→$t, publisher→$p}
where $y > 1995
construct /result/recentbook/{title←$t, publisher←$p}

In the above query, the *construct* clause is an XTree expression, which defines the result format: under the root *result*, each *recentbook* element will store the title and publisher of that book.

Example 10. For each book that has more than three authors, list its title and the first two authors, and order the result by the titles.

query /bib/book/{title→$t, author→{$a}}
where {$a}.count() > 3
order by $t
construct /result/book/{title←$t, author←{$a}[1-2]}

4 Algorithms to Tranform XTree Query to XQuery

We have seen that XTree is more compact and convenient than XPath, however, we want to transform queries based on XTree expressions to standard XQuery queries, to make them executable by existing XQuery parsers. In this section, we will present two algorithms, the first one is to transform an XTree expression in the querying part to a set of XPath expressions, and the second one is to transform an XTree expression in the result construction part to some nested XQuery expressions.

Before introducing the algorithms, we will make the following definitions.

Definition 5. Defintion 1 defines the *associated path*. Function *path($var)* returns the associated path of single-valued variable *$var*, *path({$var})* returns the associated path of list-valued variable {$var}.

For example, for XTree expression */bib/book/{title→$t, author→{$a}}, path($t) = /bib/book/title*, *path({$a}) = /bib/book/author*.

Definition 6. The *relative path* of $path_1$ with regard to $path_2$ is the path that starts from the endpoint of $path_2$ and ends at the endpoint of $path_1$. Function *relaPath($path_1$, $path_2$)* returns the relative path of $path_1$ with regard to $path_2$. It can be evaluated by a prefix elimination of $path_2$ in $path_1$.

For example, *relaPath(/a/b/c/d, /a/b) = c/d, relaPath(/a/b, /a/b) = null*

Definition 7. Variable *$a* is the *nearest ancestor variable* of variable *$b* if *$a* is an ancestor variable of *$b*, and no other ancestor variables of *$b* are defined between *path($a)* and *path($b)*.

For example, in XTree expression */bib/book→$b/{title→$t, author/last→$last}*, *$b* is the nearest ancestor variable of *$last*.

4.1 Transformation Algorithm for Querying Part

Transforming an XTree expression in the querying part to a set of XPath expressions is not just extracting each path associated with a variable to be an XPath expression, because variables may correlate to each other by some common ancestors, thus we need to use such common ancestors to constrain the descendent variables, and define them correctly.

It is very easy to get these common ancestors, by just analyzing the textual XTree expression itself. The paths just before every pair of curly braces for branching (not for list-valued variables) are the common ancestors we want, because the pair of curly braces implies that all the enclosed sibling branches are interested by the user, no matter the sibling branches will head to some variable bindings or some constraints.

Fig. 3 gives the pseudo code of the algorithm that transforms an XTree expression in the querying part to a set of XPath expressions.

```
Process the textual XTree expression from left to right
for each node traversed {
  if it is bound to a single-valued variable $svar {
    find its nearest ancestor variable $ancvar (or {$ancvar})
    if no such $ancvar found, output XPath: For $svar in path($svar)
    else output XPath: For $svar in $ancvar/relaPath(path($svar), path($ancvar))
  }
  else if it is bound to a list-valued variable {$lvar} {
    find its nearest ancestor variable $ancvar (or {$ancvar})
    if no such $ancvar found, output XPath: Let $lvar := path({$lvar})
    else output XPath: Let $lvar := $ancvar/relaPath(path({$lvar}), path($ancvar))
  }
  else if it is directly followed by a pair of curly braces for branching, and it is not the root {
    assign a single-valued variable $new_svar to this node
    find its nearest ancestor variable $ancvar (or {$ancvar})
    if no such $ancvar found, output XPath: For $new_svar in path($new_svar)
    else output XPath:
      For $new_svar in $ancvar/relaPath(path($new_svar), path($ancvar))
  }
}
```

Fig. 3. Algorithm to transform an XTree expression in querying part

The main idea of this algorithm is that we find all the common ancestors of variables, except the root (since an XML document only has one root node), and assign single-valued variables on them if they are not bound to variables originally. Then we translate each single-valued variable to be an XPath expression in a *for* clause, and translate each list-valued variable to be an XPath expression in a *let* clause. For each variable, if it has a nearest ancestor variable, we will output its XPath expression to be the relative path from its nearest ancestor variable; otherwise we will output its XPath expression to be the absolute path from the root of the document.

Note that in the above algorithm, whenever we encounter a list-valued variable *{$lvar}*, we will just use its inner name *$lvar* (without curly braces *{ }*) in the output, because in XQuery a variable defined by a *let* clause does not have curly braces in its name. Also, since we process the XTree expression in a left-to-right manner, after applying the algorithm to an XTree expression, the output paths will be in depth-first order of the XTree.

4.2 Transformation Algorithm for Result Construction Part

Transforming an XTree expression in the result construction part to some XQuery expressions is more complicated, since we will often encounter nested sub-queries in XQuery. Also, if the node name to get the variable value is different from the node name where the variable was bound in the querying part, it will be difficult to handle. Fig. 4 gives the pseudo code of the algorithm that transforms an XTree expression in the result construction part to some XQuery statements.

```
String $begin := "", String $end := ""
for ($step_i$) {          //execute step by step, suppose $step_i$ concerns $node_i$
  if $step_i$ has no "←" symbol {
    if it has compatible structure with some lowest common ancestor $anc in last algorithm
      $begin := $begin + "{" + the XPath expression of $anc from last algorithm
      $begin := $begin + "return <node_i>"
      $end := "</node_i>" + $end
    else              //it is the first several levels in result structure, e.g., root of result
      $begin := $begin + "<node_i>", $end = "</node_i>" + $end
  }
  else { //suppose the expression of current step is: $expr_i := node_i←$var_i or node_i←{$var_i}
    $begin := $begin + "{" + the XPath expressions of variable $var_i or {$var_i}
    $begin := $begin + "return " + translate($expr_i) + "}"
    if next step is descendent of current step, or this is the last step (no next step)
      $end := "</node_i> }" + $end
    else if next step is sibling of current step
      $begin := $begin + "</node_i> }"
    else //next step is sibling of some $step_j$ (which concerns $node_j$) processed before
      //let string $end_j be the substring of $end from beginning till "</node_j> }"
      $begin := $begin + "</node_i> }"
      $begin := $begin + $end_j, $end := $end - $end_j
  }
}
output $begin + $end
```

Fig. 4. Algorithm to transform an XTree expression in result construction part

The main idea of this algorithm is that we process the XTree expression in result construction part step by step. We will find the corresponding XPath expressions of each variable in the output of last algorithm, and use curly braces *{ }* to form sub-query blocks according to the structure of the XTree expression in *construct* clause.

Function ***translate($expr)*** in Fig. 5 translates a value substitution expression to be a ***return*** clause in XQuery, depending on whether the node name for value substitution is the same as the node name where the variable was bound in querying part. If the node name remains the same, it just puts the value of the variable at the place. Otherwise if the node name is changed, it will get the inner structure (subelements, attributes, text fields, etc) of the node and put their values enclosed by the new node name.

```
define function translate($expr) {
    case 1. $expr is of format: element ← $var {
        if element is the same name as the node where $var was bound in querying part
            return "{$var}"
        else                //name changed
            return "<element> {$var/*} {$var/@*} {$var/text()} </element>"
    }
    case 2. $expr is of format: element ← {$var} {
        if element is the same name as the node where {$var} was bound in querying part
            return "{$var}"
        else
            return "{ for $x in $var
                      return <element> {$x/*} {$x/@*} {$x/text()} </element> }"
    }
    case 3. $expr is of format: @attr ← $var {
        if attr is the same name as the node where $var was bound in querying part
            return "{$var}"
        else                //suppose the parent element name of $var is elem
            return "<elem attr = {string($var)}>"
    }
}
```

Fig. 5. Function *translate*

4.3 An Example

To illustrate how our algorithms work, consider the following example:

Example 11. Suppose we want to list the books and journals. For each book, we rename its *title* to be *name*, add an element *authors* that consists of all the authors of this book, add an attribute *count* in *authors* which counts the number of authors, and rename element *author* to *au*. For each editor of a journal, we put its first name before last name.

query /bib/{book/{title→$t, author→{$a}},
journal/{title→$jt, editor/{last→$last, first→$first}}}
construct /result/{book/{name←$t, authors/{@count←{$a}.count( ), au←{$a}},
journal/{title←$jt, editor/{first←$first, last←$last}}}

For the XTree expression in the querying part, by applying the first algorithm, we process it from left to right. The algorithm will output an XPath expression for each node bound to a variable and each branch node.

First we reach the node *book*, which is followed by a branch, thus we assign a new single-valued variable *$b* to it, and output a *for* clause: *for $b in /bib/book*.

Next we reach the node *title*, which is bound to a single-valued variable *$t*, and we find that it has *$b* as its nearest ancestor variable, so we will output its XPath expression with reference to *$b*: *for $t in $b/title*.

By continuing such procedure, finally we will have the following output:

```
for $b in /bib/book
for $t in $b/title
let $a := $b/author
for $j in /bib/journal
for $jt in $j/title
for $e in $j/editor
for $last in $e/last
for $first in $e/first
```

For the result construction part, by applying the second algorithm, we will get the following XQuery (detailed execution omitted) as in Fig. 6.

```
<result> {
   for $b in /bib/book
   return <book> {
      for $t in $b/title
      return <name> {$t/*} {$t/@*} {$t/text()} </name>
   }
   {
      let $a := $b/author
      return <authors count={count($a)}> {
         for $x in $a
         return <au> {$x/*} {$x/@*} {$x/text()} </au>
      }
      </authors>
   }
   </book>
   }
```

```
{
   for $j in /bib/journal
   return <journal> {
      for $jt in $j/title
      return {$jt}
   }
   {
      for $e in $j/editor
      return <editor> {
         for $first in $e/first
         return {$first}
      }
      {
         for $last in $e/last
         return {$last}
      } </editor>
   } </journal>
} </result>
```

Fig. 6. Result XQuery of Example 11

5 Conclusion

In this paper, we have illustrated some problems with XPath, and proposed our XTree which is a generalization of XPath. XTree has a tree structure, which is more compact and convenient to use than XPath. For the queries based on XTree expressions, in the querying part, multiple variables can be defined in one XTree expression; in the result construction part, a user can just write one XTree expression to define the result format. The separation of querying part and result construction part effectively avoids nested structure in the query, and makes the whole query easy to read and understand. In XTree expressions, list-valued variables are explicitly indicated, and their values are uniquely determined. Some natural built-in functions are defined to manipulate list-valued variables in an object-oriented fashion.

To utilize the current XQuery parsers, we have also designed two algorithms that convert a query based on XTree expressions to a standard XQuery query. The first algorithm transforms an XTree expression in the querying part to a set of XPath expressions, and the second algorithm transforms an XTree expression in the result construction part to some nested XQuery expressions.

For the future research, we would like to extend XTree by adding more useful features, such as negation, group by, join, etc. Currently in XQuery, grouping and join are done by nested sub-queries, which are very inefficient. We will investigate how to integrate these features into XTree expressions. Also we would like to implement an XTree query parser so that queries based on XTree expressions can be executed directly, instead of translating to XQuery queries. The querying evaluation will be more efficient on this approach, since we will have a global view of the whole query tree.

References

1. S.Abiteboul, D.Quass, J.McHugh, J.Widom, and J.L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal of Digital Library* 1(1):68-99, 1997.
2. S.Ceri, S.Comai, E.Damiani, P.Fraternali, S.Paraboschi, and L.Tanca. XML-GL: a Graphical Language for Querying and Restructuring WWW data. In *Proceedings of the 8th International World Wide Web Conference*, Toronto, Canada, 1999.
3. S.Cluet and J.Simeon. YATL: a Functional and Declarative Language for XML. Draft manuscript, March 2000.
4. H.Hosoya and B.Pierce. XDuce: A Typed XML Processing Language (Preliminary Report). In *Proceedings of WebDB Workshop*, 2000.
5. M.Liu and T.W.Ling. Towards Declarative XML Querying. In *Proceedings of WISE 2002*, 127-138, Singapore, 2002.
6. P.Chippimolchai, V.Wuwongse and C.Anutariya. Semantic Query Formulation and Evaluation for XML Databases. In *Proceedings of WISE 2002*, 205-214, Singapore, 2002.
7. D.Chamberlin, P. Fankhauser, M.Marchiori, and J.Robie. XML Query Requirements. W3C Working Draft, In http://www.w3.org/TR/xquery-requirements/, June 2003.
8. J. Clark and S.DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation, In http://www.w3.org/TR/xpath, November 2001.
9. D.Chamberlin, D.Florescu, J.Robie, J.Simon, and M.Stefanescu. XQuery 1.0: A Query Language for XML. W3C Working Draft, In http://www.w3.org/TR/xquery/, May 2003.
10. J.Robie, J.Lapp, and D.Schach. XML Query Language (XQL). In http://www.w3.org/TandS/QL/QL98/pp/xql.html, 1998.
11. A. Deutsch, M.Fernandez, D.Florescu, A.Levy, and D.Suciu. XML-QL: A Query Language for XML. In http://www.w3.org/TR/NOTE-xml-ql/, August 1998.
12. J.Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, In http://www.w3.org/TR/xslt, November 1999.