A PROTOTYPE FOR TRANSLATING XSLT INTO XQUERY

Ralf Bettentrupp
University of Paderborn, Faculty 5, Fürstenallee 11, D-33102 Paderborn, Germany

Sven Groppe, Jinghua Groppe
Digital Enterprise Research Institute (DERI), University of Innsbruck, Institute of Computer Science, AT-6020 Innsbruck

Stefan Böttcher
University of Paderborn, Faculty 5, Fürstenallee 11, D-33102 Paderborn, Germany

Le Gruenwald
University of Oklahoma, School of Computer Science, Norman, Oklahoma 73019, U.S.A

Keywords: XML, XSLT, XQuery, Source-to-Source Translation.

Abstract: XSLT and XQuery are the languages developed by the W3C for transforming and querying XML data. XSLT and XQuery have the same expressive power and can be indeed translated into each other. In this paper, we show how to translate XSLT stylesheets into equivalent XQuery expressions. We especially investigate how to simulate the match test of XSLT templates by two different approaches which use reverse patterns or match node sets. We then present a performance analysis that compares the execution times of the translation, XSLT stylesheets and their equivalent XQuery expressions using various current XSLT processors and XQuery evaluators.

1 INTRODUCTION

XSLT (W3C, 1999b) and XQuery (W3C, 2005) are both languages developed for transforming and querying XML documents. XSLT and XQuery have the same expressive power. In this paper, we show how to translate XSLT stylesheets into equivalent XQuery expressions.

Many commercial as well as freely available products support the evaluation of XQuery expressions, but do not support the XSLT language. Examples include Tamino XML Server (Software AG, 2004), Microsoft SQL Server 2005 Express (Microsoft, 2004) and Qizx (Franc, 2004). A translation module from XSLT stylesheets into XQuery expressions can make the XSLT language available for these products.

Another usage scenario is the migration of sub-systems of legacy systems from their current language, XSLT, to the new language, XQuery. Then a translation module can be used to translate the old XSLT stylesheets so that the translated XQuery expressions can be applied instead. Note that XSLT has been used in many companies for a longer time than XQuery; therefore many applications already use XSLT. Furthermore, many XSLT stylesheets for different purposes can be found on the web, but the new XQuery technology becomes more and more important in the context of XML databases and XML enabled databases. Whenever an application requires concepts primarily supported by an XML database system or an XML enabled database system (such as the ACID properties, improved query processing or improved security), most of these database systems will require the application to use the XQuery language as the query language. Again, our contribution enables the user to alternatively formulate queries in the XSLT language and, afterwards, apply our proposed XSLT to XQuery translator to obtain equivalent XQuery expressions.

Bettentrupp R., Groppe S., Groppe J., Böttcher S. and Gruenwald L. (2006).
A PROTOTYPE FOR TRANSLATING XSLT INTO XQUERY.
In *Proceedings of the Eighth International Conference on Enterprise Information Systems - DISI*, pages 22-29
DOI: 10.5220/0002442100220029

2 TRANSLATION APPROACH

Due to space limitations, we do not describe XSLT and XQuery in detail here, but refer interested readers to their specifications published in (W3C, 1999b) and (W3C, 2005) respectively.

2.1 Differences Between XSLT and XQuery

XSLT 2.0 and XQuery 1.0 are both based on the XPath data model and both embed XPath as the path language for determining XML node sets. Therefore, a majority of the XSLT language constructs can be translated into XQuery language constructs. For example, `xsl:for-each` has similar functionality as `for`, `xsl:if` has similar functionality as `where` and `xsl:sort` has similar functionality as `order by`.

However, XSLT uses a template model, where each template contains a pattern in form of an XPath expression. A template model is not supported by XQuery and must be simulated in the translated XQuery expression.

XSLT and XQuery deal with parameters of functions in a different way: Whereas XSLT binds parameters of calls of functions and templates by parameter names, XQuery binds parameters in function calls by parameter positions. Thus, we have to simulate named parameters using a data structure containing the names and the values of the parameters.

The given mode in a template definition and in calls of templates defines in XSLT which templates can be called.

Many complex XSLT instructions are not supported by XQuery and must be simulated by user-defined functions of a runtime library.

The translated XQuery expression has to simulate different functionalities of XSLT. An example of an XSLT stylesheet and its translated XQuery expression is given in Section 2.2. The translated XQuery expression has to simulate the *template selection process*. For the template selection process, we present two different approaches: the *match node set approach* (see Section 2.3.1) and the *reverse pattern approach* (see Section 2.3.2). Besides simple XSLT instructions that can be easily translated into an XQuery expression, some complex XSLT instructions do not have corresponding functions with the same functionality in XQuery. In Section 2.4, we outline how to use an XQuery runtime library of those functions that simulate these complex XSLT instructions. The overall translation process is described in Section 2.5.

2.2 Translation Example

```
<xsl:stylesheet>
<xsl:template match="table">
  <table> <xsl:apply-templates select="row">
            <xsl:sort select="firstname"/>
          </xsl:apply-templates> </table>
</xsl:template>
<xsl:template match="*">
  <xsl:copy><xsl:apply-templates/></xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

Figure 1: Example XSLT stylesheet stringsort.xslt of the XSLTMark benchmark.

The XSLT stylesheet of Figure 1, which contains an XSLT stylesheet of the XSLTMark benchmark (Developer, 2005), is translated into the XQuery expression of Figure 2 using the match node sets approach described in Section 2.3.1.

```
declare variable $doc_node:=fn:doc("E:/db100.xml");
declare variable $matchBuiltIn1 as node()* :=
  $doc_node |
  $doc_node/descendant-or-self::node()/child::*;
declare variable $matchBuiltIn2 as node()* :=
  $doc_node/descendant-or-self::node()/text()|
  $doc_node/descendant-or-self::node()/attribute::node();
declare variable $match1 as node()* := $doc_node/
  descendant-or-self::node()(/self::node()|
  /attribute::node()|/namespace::node())/table;
declare variable $match2 as node()* := $doc_node/
  descendant-or-self::node()(/self::node()|
  /attribute::node()|/namespace::node())/*;
declare variable $noValue := <root>NOVALUE</root>;

declare function local:paramTest($name as item()*, $select
as item()*) as item()* {
  if(fn:empty($name)) then $select
  else if(fn:compare(fn:string($name),xs:string("NOVALUE"))
          =0) then () else $name };

declare function local:copy($n as node(), $value as
item()*)as item()*{
  if($n instance of element()) then
    element {name($n)} { $value }
  else if($n instance of attribute()) then
    attribute {name($n)} {xs:string($n)}
  else if($n instance of text()) then xs:string($n)
  else if($n instance of comment()) then
         comment{xs:string($n)}
  else if($n instance of processing-instruction())
       then processing-instruction {name($n)}
          {xs:string($n)} else () };

declare function local:builtInTemplate1($t as node(),
$param as item()*) as item()* {
 let $gerg:=local:apply_templates($t/child::*,$noValue)
 return $gerg };

declare function local:builtInTemplate2($t as node(),
$param as item()*) as item()* {
 let $gerg := xs:string($t) return $gerg };

declare function local:template1($t as node(), $param as
item()*) as item()* {
 let $zerg1 := element table{
  let $erg1 := $t/(row)
  let $erg21 := for $t in $erg1 order by $t/firstname
                ascending return $t
  let $zerg1:=local:apply_templates($erg21,$noValue)
  let $gerg := ($zerg1) return $gerg }
 let $gerg := ($zerg1) return $gerg };

declare function local:template2($t as node(), $param as
item()*) as item()* {
 let $zerg1 :=
  let $erg1 := $t/(child::node())
  let $erg21 := $erg1
  let $zerg1:=local:apply_templates($erg21,$noValue)
  let $gerg := ($zerg1)
  return local:copy($t, $gerg)
 let $gerg := ($zerg1) return $gerg };
```

```
declare function local:apply_templates($n as node()*,
$param as item()*)as item()* {
  for $t in $n
  return if($t intersect $match1) then
           local:template1($t, $param)
         else if($t intersect $match2) then
                local:template2($t, $param)
         else if($t intersect $matchBuiltIn1) then
                local:builtInTemplate1($t, $param)
         else if($t intersect $matchBuiltIn2) then
                local:builtInTemplate2($t, $param)
         else () };

let $doc:=$doc_node
return local:apply_templates($doc,$noValue)
```

Figure 2: Translated XQuery expression of the XSLT stylesheet of Figure 1 using the match node set approach (see Section 2.3.1).

The function `local:paramTest` is used for the simulation of parameters in an XSLT stylesheet. The function `local:copy` is a function of the runtime library that simulates the `<xsl:copy>` XSLT instruction. The functions `local:builtInTemplate1` and `local:builtInTemplate2` simulate the built-in templates of XSLT. `local:template1` and `local:template2` contain the translation of the user-defined templates of Figure 1. `local:apply_templates` simulates the `<xsl:apply-templates>` XSLT instruction. The translated XQuery expression does not consider different modes for the call of templates as the original XSLT stylesheet does not use different modes. In general, the function `local:apply_templates` must have an additional `mode` parameter and must consider the value of `mode` and the modes of the XSLT templates for calling a template.

When using the reverse pattern approach, the translated XQuery expression does not contain the declaration of the variables `$matchBuiltIn1`, `$matchBuiltIn2`, `$match1` and `$match2`. Furthermore, we translate the function `local:apply_templates` into the one given in Figure 3 instead of the one given in Figure 2.

```
declare function local:apply_templates($n as node()*,
$param as item()*)as item()* {
  for $t in $n return
   if($t[self::table[self instance of
              element()*]/parent::node()])
            then local:template1($t, $param)
   else if($t[self::*[self instance of
          element()*]/parent::node()])
        then local:template2($t, $param)
   else if($t is root($t) or $t/self::element())
        then local:builtInTemplate1($t, $param)
   else if($t/self::text() or $t/self::attribute())
        then local:builtInTemplate2($t, $param) else () };
```

Figure 3: Function local:apply_templates when using the reverse pattern approach (see Section 2.3.2).

2.3 Template Selection Process

XSLT uses a template model, where each template contains a pattern in form of an XPath expression. Whenever a current input XML node fulfills the pattern of a template, the template is executed. An XSLT processor starts the transformation of an input XML document with the document node assigned to the current input XML node. The templates are again called when the XSLT processor executes the `<xsl:apply-templates select=I/>` instructions, which first select a node set `I` and then call the templates for all nodes in `I`.

In this section, we present how a template is selected for execution.

After initialization, the XSLT processor starts processing the XSLT stylesheet by applying the templates to the document node `/`. We simulate the status of the XSLT processor's input by using two variables. The first variable, `$n`, represents the current input XML node set of the XSLT processor and is initialized with a set containing the document node `/`. The second variable, `$t`, represents the current input XML node of the XSLT processor. We use `$t` in order to iterate over the current input node set `$n`. Depending on `$t`, we start the template selection process, the code of which is generated in the XQuery function `local:apply_templates` as follows.

We sort the templates according to their priority (either explicit priority or computed default priority) and import precedence (see (W3C, 1999b)), where the built-in templates of XSLT are the templates with the lowest priority. The selection process of the templates is then coded in XQuery by first executing the match test of the template with the highest priority and, in the case of success, executing the corresponding translated template. In the case of no success, the match test of the next template in the sorted list of templates is executed and, in the case of success, the corresponding translated template is processed. The translation generates code of the template selection analogously for all other templates in the list in the given order.

XQuery does not support checking the XPath patterns of XSLT. Thus, we have to simulate the test whether a *pattern* `E` *matches an XML node* in XQuery. The original definition in the XSLT specification (W3C, 1999b) is as follows:

Definition 1: A pattern is defined to *match* a node if and only if there is possible context such that when the pattern is evaluated as an expression with that context, the node is a member of the resulting node-set. When a node is being matched, the possible contexts have a context node that is the node being matched or any ancestor of that node, and a context node list containing just the context node.

We present two different approaches, the match node sets approach (see Section 2.3.1) and the reverse pattern approach (see Section 2.3.2), for the simulation of checking XPath patterns of XSLT in

XQuery. Furthermore, we describe in Section 2.3.3 how to simulate the use of parameters in the call of templates.

2.3.1 The Match Node Set Approach

For the test whether a template `t` matches an input XML node `$c`, the match node set approach checks whether `$c` is contained in a pre-computed node set, the *match node set* of `t`. The match node set of `t` contains all nodes that could be matched by `t`.

Definition 2: An XPath expression `I` can be divided into a *relative part* `rp(I)` and an *absolute part* `ap(I)` (both of which may be empty) in such a way that `rp(I)` contains a relative path expression, `ap(I)` contains an absolute path expression, and the union of `ap(I)` and `rp(I)`, i.e. `ap(I)|rp(I)`, is equivalent to `I`. This means that applying `I` and applying `ap(I)|rp(I)` will return the same node set for all XML documents and for all context nodes in the current XML document.

Example 1: Let `I` be `(/child::a|child::b) /attribute::c`. The relative part of `I` is `rp(I)=child::b/attribute::c`, the absolute part of `I` is `ap(I)=/child::a/attribute::c`.

Definition 3: The *match node set* of a template `<xsl:template match=M>` are those XML nodes, which are matched by `M`.

Proposition 1: Given a template `<xsl:template match=M>`. If the absolute part of `M` and the relative part of `M` are non-empty, i.e. `ap(M)≠{}` and `rp(M)≠{}`, the match node set of the template can be computed by applying the XPath query `ap(M)|/descendant-or-self::node()(/self::node()|/attribute::node()|/namespace::node())/rp(M).`

If ap(M)={} and rp(M)≠{}, the match node set of the template can be computed by applying the XPath query `/descendant-or-self::node()(/self::node()|/attribute::node()|/namespace::node())/rp(M).`

If `ap(M)≠{}` and `rp(M)={}`, the match node set of the template can be computed by applying the XPath query `ap(M)`. If `ap(M)={}` and `rp(M)={}`, the match node set of the template is an empty set.

Proof of Proposition 1: The XPath expression `/descendant-or-self::node()(/self::node()|/attribute::node()|/namespace::node())` returns all XML nodes of an input XML document. All XML nodes, which are matched by `M`, are the union (expressed by using the operator "|") of the absolute part of `M`, `ap(M)`, and of those XML nodes which are returned from the evaluation of `M` relative to each XML node. □

In the case that we only have to check whether patterns match XML nodes of the input XML document, we declare a variable for the match node set of each template so that each match node set is only computed once in the XQuery expression. In the case that we have to check whether patterns match XML nodes of computed variables, we must compute the match node set of a variable after its computation. Furthermore, in order to check whether a current input XML node `$c` is in a match node set `$MN`, we use the XPath expression `$c intersect $MN`, which returns the node $c if $c is in $MN.

2.3.2 The Reverse Pattern Approach

For the test whether a template `t` with a `match` attribute `E` matches a current input XML node `$c`, the reverse pattern approach checks whether $c[E^{-1}]≠∅, where E^{-1} is the *reverse pattern* of `E`.

We present an extended variant of the approaches in (Moerkotte, 2002) and in (Fokoue, 2005) for a superset of the XPath patterns of XSLT. In comparison to the approaches presented in (Moerkotte, 2002) and (Fokoue, 2005), we present the general rules for generating the reverse pattern.

To determine the reverse pattern of a given XPath expression, we first define the reverse axes of an XPath axis as shown in Figure 4.

Definition 4: The *reverse axes* of a given XPath axis are defined in the middle column of Figure 4.

Axis A	Reverse Axes of A	Additional Test
ancestor	1) descendant 2) descendant-or-self::node()/attribute 3) descendant-or-self::node()/namespace	
ancestor-or-self	1) descendant-or-self 2) descendant-or-self::node()/attribute 3) descendant-or-self::node()/namespace	
attribute	parent	[self instance of attribute()*]
child	parent	[self instance of element()*]
descendant	ancestor	
descendant-or-self	ancestor-or-self	
following	preceding	
following-sibling	preceding-sibling	
namespace	parent	[not (self instance of element()*) and not(self instance of attribute()*)]
parent	1) child 2) attribute 3) namespace	
preceding	following	
preceding-sibling	following-sibling	
self	self	

Figure 4: Reverse axes and additional test of an XPath axis.

Note that the parent of an attribute or a namespace node is its element node, but an attribute or namespace node is not a child of its element node. Therefore, attribute nodes and namespace nodes cannot be accessed by the `child` or `descendant` axes, and also not by the `descendant-or-self` axis if the attribute node or namespace node is not the current context node. An attribute node can only be accessed by the `attribute` axis and a namespace node only by the `namespace` axis. Thus, there is more than one reverse axis of the `ancestor`, `ancestor-or-self` or `parent` axes (see Figure 4).

The reverse axis of the `attribute` axis, of the `child` axis and of the `namespace` axis is the `parent` axis, which does not differ for attribute, namespace and other nodes (in comparison to the original axis). Therefore, we use an *additional test* (see Definition 5) in the definition of the reverse pattern (see Definition 6) to distinguish different node types.

Definition 5: The *additional test* of a given XPath axis is defined in the right column of Figure 4.

Definition 6: The *reverse pattern* of a given XPath expression is computed as follows: At first, we transform the XPath expression into its long form. If there are disjunctions ("|") in the XPath expression outside of a filter expression, then we factor out the disjunctions and reverse each expression of the disjunctions separately. The whole reverse pattern is the disjunction of all separately reversed expressions. Without disjunctions, a relative XPath expression $E_{relative}$ has the form

$$axis_1::test_1[F_{11}]\ldots[F_{1n_1}]/axis_2::test_2[F_{21}]\ldots[F_{2n_2}]/\ldots/ axis_m::test_m[F_{m1}]\ldots[F_{mn_m}],$$

and an absolute XPath expression $E_{absolute}$ has the form

$$/axis_1::test_1[F_{11}]\ldots[F_{1n_1}]/axis_2::test_2[F_{21}]\ldots[F_{2n_2}]/\ldots/ axis_m::test_m[F_{m1}]\ldots[F_{mn_m}]$$

where $axis_i$ are XPath axes, $test_i$ are node tests and F_{ij} are filter expressions. The reverse pattern of $E_{relative}$ and of $E_{absolute}$ is

$$\begin{aligned} &self::test_m[F_{m1}]\ldots[F_{mn_m}]T_m/(raxis_{m1}::test_{m-1}|\ldots| \\ &\qquad raxis_{mp_m}::test_{m-1})[F_{(m-1)1}]\ldots[F_{(m-1)n_{m-1}}]T_{m-1}/\ldots/ \\ &(raxis_{21}::test_1|\ldots|raxis_{2p_2}::test_1)[F_{11}]\ldots[F_{1n_1}]\ T_1/ \\ &(raxis_{11}::node()|\ldots|raxis_{1p_1}::node())\ T_{root}, \end{aligned}$$

where T_{root} is `[self::node() is root()]` for $E_{absolute}$ and T_{root} is the empty expression for $E_{relative}$, $raxis_{i1}$ … $raxis_{ip_i}$ are the reverse axes of $axis_i$, and T_i is the additional test of $axis_i$ as outlined in Figure 4, or T_i is the empty expression if there is no additional test of $axis_i$.

2.3.3 Simulating the Use of Parameters

Whereas XSLT binds parameters of calls of functions and of templates by parameter names, XQuery binds parameters in function calls by parameter positions. Furthermore, XQuery functions and, especially the function `local:apply_templates`, do not support an arbitrary number of parameters. Thus, we have to simulate named parameters using a data structure containing the names and the values of the parameters. For an example, for the template call

```
<xsl:apply-templates select="*">
  <xsl:with-param name="a1" select="$a"/>
  <xsl:with-param   name="a2"><xsl:copy-of   select="$a"/>
  </xsl:with-param>
</xsl:apply-templates>
```

we use the following data structure for the simulation of parameters

```
<root> <a1> value of $a </a1><a2> value of $a </a2> </root>
```

This data structure is generated by the following translated XQuery expression:

```
let $a :=  $t/(*) let $erg1 := $t/(*) let $erg21 := $erg1
let $newParam :=  element root {
  element a1 { $a }, element a2 { let $erg1 := $a
    let $zerg1 := local:copy_of($erg1)
    let $gerg :=  ($zerg1) return $gerg } }
let $zerg1 :=(local:apply_templates($erg21,$newParam))
return $zerg1
```

In general, the parameters are stored in an XML tree with the root element `<root>`. The parameters are children of the root element containing the value of the parameter. The data structure used is as follows

```
<root> <PARAM-NAME_1>PARAM_1</PARAM-NAME_1>
     … <PARAM-NAME_n>PARAM_n</PARAM-NAME_n> </root>
```

where `PARAM-NAME_i` represents the name of the `i`-th parameter and `PARAM_i` represents its value. In order to access the parameters, we use the function `local:paramTest(…)` given in Figure 2.

Due to the simplicity of the translation approach, the simulation of parameters for templates does not work correctly in rare cases. Let us consider the parameter `a1` in the example above. In XSLT, the identities of the XML nodes of the variable `$a` remain the same, but the identities of the copied XML nodes in the data structure used for the translated XQuery expression differ from the ones in `$a`. Thus, we do not retrieve the same result when the XML nodes of the parameter `a1` are compared with the XML nodes of `$a` by the XPath operators `is`, `<<` or `>>`. In order to avoid this problem, we propose three different strategies. First, the function `local:apply_templates` could have a large list of parameters to represent all possible parameters in the XSLT stylesheet. This solution does not work if the parameter name depends on the input. Second, the

function `local:apply_templates` could be inlined so that functions that simulate the templates with different numbers of parameters could be called. In general, this solution does not work either if the parameter name depends on the input. Third, the identity of an XML node can be stored in an extra attribute that is copied in the case of parameter `a1` similar to the pre-processing step and post-processing step discussed in (Klein et al., 2005) and (Lechner et al., 2001) for XQuery to XSLT translation. The XPath operators `is`, `<<` or `>>` must then be replaced by the ones that operate on this extra identity attribute.

2.4 Runtime Library

The proposed translator uses a runtime library of XQuery functions, which simulate certain XSLT instructions. The runtime library includes the functions simulating `<xsl:copy>` and `<xsl:copy-of>`, which are given in Figure 2 and Figure 5, respectively. The complex XSLT instructions `<xsl:number>` and `<xsl:message>` are also candidates for functions of the runtime library, which have not yet been implemented in our prototype.

```
declare function local:copy_of($n as node()*)as item()*{
  for $t in $n return
  if($t instance of element())then
    let $next :=($t/child::node()|$t/attribute::node())
    let $new := element {name($t)}{local:copy_of($next)}
    return $new
  else if($t instance of attribute()) then
    let $new := attribute {name($t)}{xs:string($t)}
    return $new
  else if($t instance of text()) then
    let $new := xs:string($t) return $new
  else if($t instance of comment())then
    let $new := comment {xs:string($t)} return $new
  else if($t instance of processing-instruction())
  then let $new := processing-instruction{name($t)}
       {xs:string($t)} return $new else () };
```

Figure 5: Function simulating <xsl:copy-of>.

2.5 Translation Process

The translation process is executed in three phases as follows. In Phase one, we parse the XSLT stylesheet in order to generate its abstract syntax tree. For an example, Figure 6 shows the abstract syntax tree of the XSLT stylesheet of Figure 1. In Phase two, the function `local:apply_templates` is generated as described in Section 2.3 using the match node sets approach (see Section 2.3.1) or using the reverse pattern approach (see Section 2.3.2). In Phase three, we apply an attribute grammar, which we do not present here due to space limitations, to the abstract syntax tree of the XSLT stylesheet. The attribute grammar describes how to transform XSLT instructions into simple XQuery statements or into a call to functions of the runtime library (see Section 2.4).

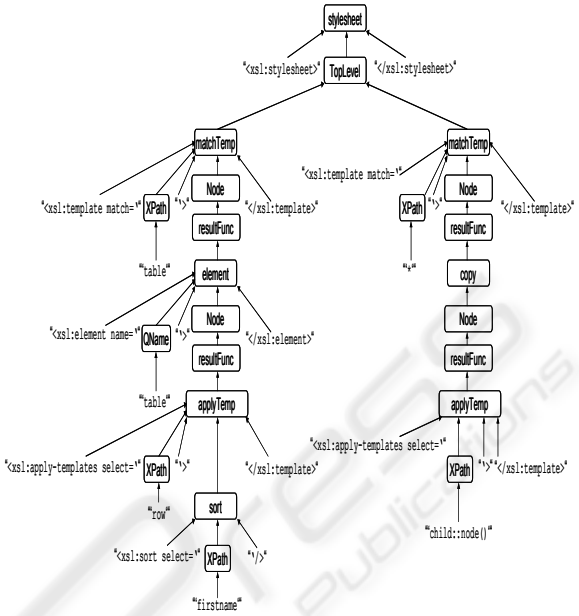

Figure 6: Abstract syntax tree of the XSLT stylesheet of Figure 1.

3 PERFORMANCE EVALUATION

This section describes the experiments that we have conducted to compare the execution times of translated XQuery expressions with the execution times of the original XSLT stylesheets.

We have run the experiments on an Intel Pentium 4 with 2 GHZ and 512 MB main memory. The system runs Windows XP and Java 1.5.

For the evaluation of XQuery expressions, we have used Qizx (Franc, 2004) and Saxon (Kay, 2004), where we have stored the output in a string. For the execution of the XSLT stylesheets, we have used Xalan (Apache Software Foundation, 2003) and Saxon (Kay, 2004). We have used the XSLT stylesheets of the XSLTMark benchmark (Developer, 2005) for our experiments.

The XSLTMark benchmark consists of 39 stylesheets, which are divided into two groups of stylesheets. The first group consists of XSLT stylesheets, each of which uses one own XML document. The XSLT stylesheets of the second group use a data set representing a database table and vary in their sizes.

We present the average execution times of ten experiments of the original XSLTMark queries using the Xalan and Saxon XSLT processor with an input stream for reading the input XML document as

input. In a variant, we first generate the DOM tree from the input XML document and use this DOM tree as input for the XSLT processors. We have measured the average execution times of ten experiments of the translated XQuery expressions (including the time used for translation) with the reverse pattern approach and with the match node sets approach using the Saxon and Qizx XQuery evaluators. We present the faster variant of the match node sets approach, where we check the intersection of the current node `$t` and of the match node set `$MN` with the following two different XPath expressions: (1) `$t intersect $MN` and (2) `some $tmp in $MN satisfies $tmp is $t`. The Qizx XQuery evaluator is faster when using the variant (1) and the Saxon XQuery evaluator is faster when using (2).

In Figure 7, we present the execution time of the XSLT stylesheets which use their own XML documents. Here, the Qizx evaluator is faster than Saxon most of the time, but processing the translated XQuery queries is slower than processing the original XSLT stylesheets in most cases. Processing the translated XQuery expression is faster than processing the original XSLT stylesheet e.g. for `html` and `priority` using the reverse pattern approach and Qizx in Figure 7. For the XSLT stylesheets with their own XML documents and their translated XQuery expressions, when using the match node sets approach, the Qizx XQuery evaluator consumes only 26% of the execution time of the Saxon XQuery evaluator, but when using the reverse pattern approach, this figure becomes 61%. The Qizx (Saxon respectively) XQuery evaluator using the match node sets approach is 8.69 times (20.66 times respectively) slower than the Qizx (Saxon respectively) XQuery evaluator using the reverse pattern approach. The XSLT processors using input streams as input consume 90% of the execution time of the XSLT processors using DOM as input. The Saxon XQuery evaluator using the reverse pattern approach is 14% slower than the Saxon XSLT processor using input streams as input.

We present the execution times of the second group of XSLT stylesheets in Figure 8 using the `db4000.xml` input XML document (with size of 785 Kilobytes). Except for rare cases (e.g. XSLTMark queries, `axis` and `metric`, in Figure 7), the reverse pattern approach is more efficient than the match node sets approach because the reverse pattern approach avoids the pre-computation of the match node sets. For the XSLT stylesheets with the input XML document `db4000.xml` and their translated XQuery expressions, the Qizx (Saxon respectively) XQuery evaluator using the match node sets approach is 41.6 times (203.36 times respectively) slower than the Qizx (Saxon respectively) XQuery evaluator using the reverse pattern approach. The XSLT processors using input streams as input consume 96% of the execution time of the XSLT processors using DOM as input. The Saxon XQuery evaluator using the reverse pattern approach is 69% slower than the Saxon XSLT processor using input streams as input.

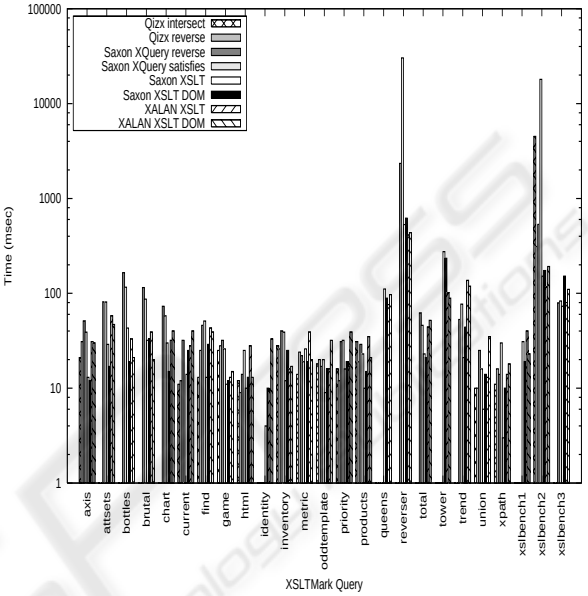

Figure 7: Execution times of XSLTMark queries with their own XML documents.

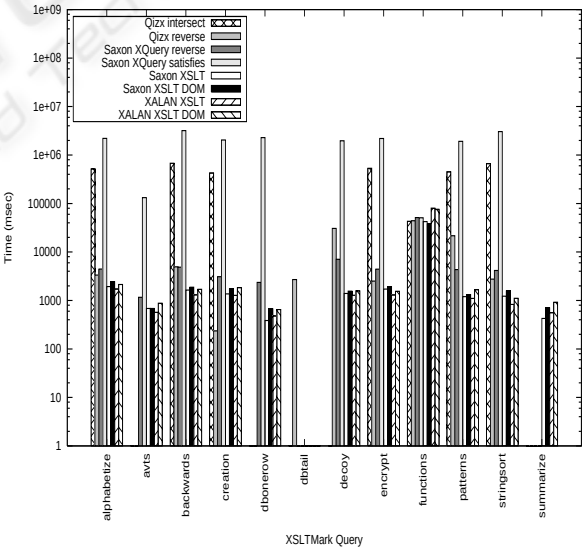

Figure 8: Execution times of XSLTMark queries with db4000.xml document.

4 RELATED WORK

There exist already contributions, which compare the languages XSLT and XQuery. (Lenz, 2004)

shows that many XQuery constructs are easily mappable to XSLT by examples, but does not provide an algorithm for translating XQuery expressions into XSLT stylesheets.

(Klein et al., 2005) and (Lechner et al., 2001) present an algorithmic approach of translating XQuery expressions into XSLT stylesheets, which is the opposite direction to our translation.

Saxon (Kay, 2005) is a processor for both, for XQuery expressions and for XSLT stylesheets, which uses a mostly common object model.

(Moerkotte, 2002) describes how XSL processing can efficiently be incorporated into database management systems. We extend the reverse pattern approach of (Moerkotte, 2002) by all axes of XPath. In comparison to (Moerkotte, 2002), we present a translation method from XSLT stylesheets into XQuery expressions and introduce the match node sets approach.

(Fokue, 2005) describes a translation from XSLT stylesheets into XQuery expressions. In comparison to (Fokue, 2005), we additionally introduce the match node set approach. (Groppe, 2005) includes a chapter dealing with the translation from XSLT stylesheets to XQuery expressions.

5 CONCLUSIONS

In this paper, we have proposed a translation process converting XSLT stylesheets to XQuery expressions. The main difficulty is to simulate the template selection of XSLT. In order to identify the template to be executed, we have presented two different methods for the simulation of the template selection process. The *match node sets approach* checks whether the current XML node is contained in a pre-computed set of XML nodes and the *reverse pattern approach* executes the reversion of the match patterns of templates. We have developed a runtime library, which contains functions in order to simulate XSLT instructions. The remaining XSLT instructions are inlined by XQuery sub-expressions.

We have carried out several experiments. In rare cases, the translated XQuery expressions using the reverse pattern approach with the XQuery evaluator Qizx are a little bit faster than the execution of the original XSLT stylesheet. Except of rare cases, the reverse pattern approach is faster than the match node sets approach for the XQuery expressions.

Therefore, we have achieved the goal to make XSLT practically usable for the broad fields of XQuery tools, XML databases and XML enabled databases, which support XQuery.

ACKNOWLEDGEMENTS

This material is based upon works supported by the EU funding under the Adaptive Services Grid project (FP6 – 004617). Furthermore, this material is based upon works supported by the Science Foundation Ireland under Grant No. SFI/02/CE1/I131. This material is based upon work supported by (while serving at) the National Science Foundation (NSF). Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

Apache Software Foundation, 2003. Xalan-Java, http://xml.apache.org/xalan-j/index.html.

Developer, 2005. XSLT Mark version 2.1.0, http://www.datapower.com/xmldev/xsltmark.html.

Franc, X., 2004. Qizx/open version 0.4p1, http://www.xfra.net/qizxopen/.

Fokoue, A., Rose, K., Siméon, J., and Villard, L., 2005. Compiling XSLT 2.0 into XQuery 1.0, *WWW 2005*, Chiba, Japan.

Groppe, S., 2005. XML Query Reformulation for XPath, XSLT and XQuery, Sierke-Verlag, ISBN 3-933893-24-0, Göttingen.

Kay, M. H., 2004. Saxon - The XSLT and XQuery Processor, http://saxon.sourceforge.net.

Klein, N., Groppe, S., Böttcher, S., and Gruenwald, L., 2005. A Prototype for Translating XQuery Expressions into XSLT Stylesheets, In *ADBIS*, Talinn, Estonia.

Lechner, S., Preuner, G., and Schrefl, M., 2001. Translating XQuery into XSLT, In *ER 2001 Workshops*, Yokohama, Japan.

Lenz, E., 2004. XQuery: Reinventing the wheel? http://www.xmlportfolio.com/xquery.html.

Microsoft, 2004. SQL Server 2005 Express, http://www.microsoft.com/sql/express.

Moerkotte, G., 2002. Incorporating XSL Processing Into Database Engines. In *VLDB*, Hong Kong, China.

Software AG, 2004. Tamino XML Server, http://www.softwareag.com/tamino/News/tamino_41.htm.

W3C, 1999a. XML Path Language (XPath) Version 1.0, W3C Recommendation, http://www.w3.org/TR/xpath/.

W3C, 2005. XQuery 1.0: An XML Query Language, W3C Working Draft.

W3C, 1999b. XSL Transformations (XSLT) Version 1.0, W3C Recommendation, http://www.w3.org/TR/1999/REC-xslt-19991116.