A Critical Analysis of XSLT Technology for XML Transformation

Gregory Sherman

Senior Technical Report

April 2008

Supervisor: Harold Boley

Introduction

The massive increase in the exchange of information that has followed the advent of the internet has led to many innovations in knowledge exchange. XML (Extensible Markup Language), a general purpose markup language, developed as a successor to SGML (Standard Generalized Markup Language), has many different applications in countless domains of knowledge. A great strength of XML is its ability to be syntactically transformed across semantically compatible domains, allowing different applications to use the same information. A member of the XSL family of languages, XSLT (XML Stylesheet Language Transformations), is a Turing complete, template-based programming language, itself written in XML, whose purpose is to transform XML information between XML languages, to text or other types of knowledge representation.

This paper will explore the uses of XSLT as template-based XML translation technology, consider the differences between the template-based and iterative styles of using XSLT, and evaluate an up and coming alternative to XSLT.

XML

In order to properly discuss XSLT, it is important to obtain a firm grasp of the concepts behind XML. XML (Extensible Markup Language) is a more restricted subset of Standard Generalized Markup Language (SGML). XML is designed to be a fully machine readable, while still being relatively human-legible. At its core, XML is purely Unicode text, and as such can be easily shared between applications, and across networks and domains. As a markup language, it does not directly provide methods for manipulating data, but is an extensible base for an infinitely diverse family of languages. Uses of XML include domain information serialization (RDF), document encoding (OOXML), vector graphics (SVG), rule exchange (RuleML/XML) and inter-application protocols (SOAP).

Anatomy of an XML document

An XML element is represented as a tag, some content contained within two angle brackets e.g. `<tag>`, followed by the element's content, and closed by a matching end tag e.g. `</tag>`.

```
<text>content</text>
```

The content of these tags can be simply text i.e. Parsed Character Data (PCDATA), other elements or (rarely) a combination of both. In addition to such content, elements may also be annotated with attributes, adding more descriptive information to the information already presented. An attribute consists of an attribute name and "=" – associated with Character Data (CDATA).

```
<tag attribute="attributevalue">content</tag>
```

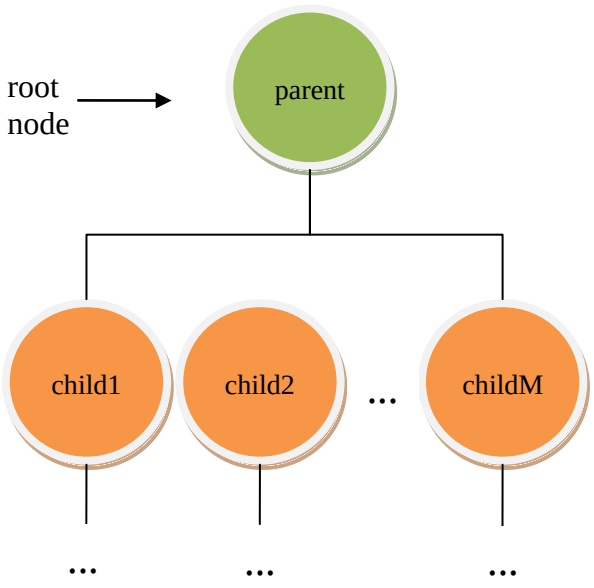
Attributes may be used to provide refined descriptions for elements, such as defining types, arities, validities etc. A single element may have multiple attributes, and for some XML languages, the represented knowledge may be entirely contained within such attributes.

XML elements may be nested inside each other, i.e., each child element of an element must be closed before that element's sibling can be defined. Elements satisfying this condition, as well as a number of other conditions, are said to be well-formed. Well-formed XML follows a tree structure, with a single parent element or root containing zero or more child elements, each of which optionally being the root of a subtree.

```
<parent>

 <child1>...</child1>
 <child2>...</child2>
  ...
 <childM>...</childM>

</parent>
```



When an XML document has neither a text value, nor children, it may be closed using compressed notation.

```
<element/> ⇔ <element></element>
```

While an XML document can only contain a single root, it may also have extra information, such as comments and processing instructions. This information exists separate from the rest of the document tree.

```
<!-- this is a comment -->
```

The following processing instruction is meant to be read by a web browser (or any other program that can use XSLT stylesheets), which will then format the document tree as specified by the stylesheet.

```
<?xml-stylesheet href="style.xsl" type="text/xml"
alternate="yes"?>
```

Processing instructions are not required to follow any standard, and will simply be ignored if the application processing the XML tree does not recognize the instruction.

Namespaces may be utilized by adding prefixes to XML elements, separated from the element by a colon.

```
<xsl:template>
```

This element belongs to the Extensible Stylesheet Language family; hence the `<template>` element of that family is prefixed with `xsl`. When namespaces are used in an XML document, it is important to define the namespace at the document root.

Examples of XML-based languages

Since the inception of XML, many languages have been developed using XML to express information and knowledge. The following languages are examples of XML use, as well as possible reasons we may want or need to transform such representations.

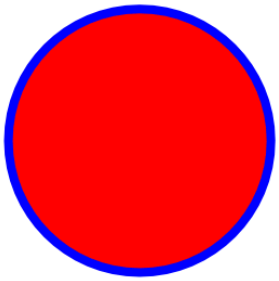
SVG

Scalable Vector Graphics (SVG) is a specification for describing two-dimensional vector graphics. Vector graphics consist of relative points and mathematical formulas for curves, so as images are scaled up or down in size, they will not lose definition. For example the following SVG code will draw a circle with a radius of 30 pixels from an origin point at x = 40 and y = 40. The resulting circle will be red, with a blue outline (stroke) of width 2.

```
<svg xmlns = "http://www.w3.org/2000/svg">

  <circle r = "30" cx= "40" cy="40"
   style ="fill: red; stroke: blue;
   stroke-width:2"
   id = "ExampleRedCircle"/>

</svg>
```



XHTML

While HTML (also a derivative of SGML) shares many characteristics with XML, it is not an XML based language, so as XML became commonplace, it was evident that an XML-based counterpart for HTML was needed, XHTML. The most striking difference between the two is that while HTML need not be well formed, it is a strict requirement for XHTML. As of the writing of this paper an XHTML2 specification is being drafted by the W3C. The following example is the above SVG image imbedded in an XHTML document, which will display a red circle in SVG compatible browsers.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Sample XHTML Document</title>
  </head>
  <body>
    <a href="http://www.example.org">
    This is a link
    </a>
    <svg xmlns = "http://www.w3.org/2000/svg">
    <circle r = "30" cx= "40" cy="40"
     style = "fill: red; stroke: blue; stroke-width:2"
     id = "ExampleRedCircle"/>
    </svg>

  </body>
</html>
```

As the World Wide Web matures, it is becoming common to see different technologies melded together to accomplish various design goals. One possible use of XSLT would be to separate embedded XML, in this case to automatically extract SVG objects from the containing XHTML.

RuleML/XML

RuleML is a markup language for publishing and sharing rule bases on the World Wide Web. Rather than a single purpose language, RuleML is a family of languages, each designed to focus on a certain area of knowledge, but still remaining interoperable with other languages in the family. This is a prime example of a language that could benefit from the ability to transform between languages. Being able to transform knowledge into this language allows users to infer and create relationships between different domains of knowledge.

The following example is an atomic formula (a fact) in Object Oriented RuleML/XML representing the SVG circle previously described:

```
<Atom>
     <Rel>object</Rel>
     <slot>
       <Ind>id</Ind>
       <Ind type="#Circle">ExampleRedCircle</Ind>
     </slot>

     <slot>
       <Ind>r</Ind>
       <Ind>30</Ind>
     </slot>

     <slot>
       <Ind>cx</Ind>
       <Ind>40</Ind>
     </slot>

     <slot>
       <Ind>cy</Ind>
       <Ind>40</Ind>
     </slot>

     <slot>
       <Ind>style</Ind>
       <Ind>"fill: red; stroke: blue; stroke-width:2"</Ind>
     </slot>

</Atom>
```

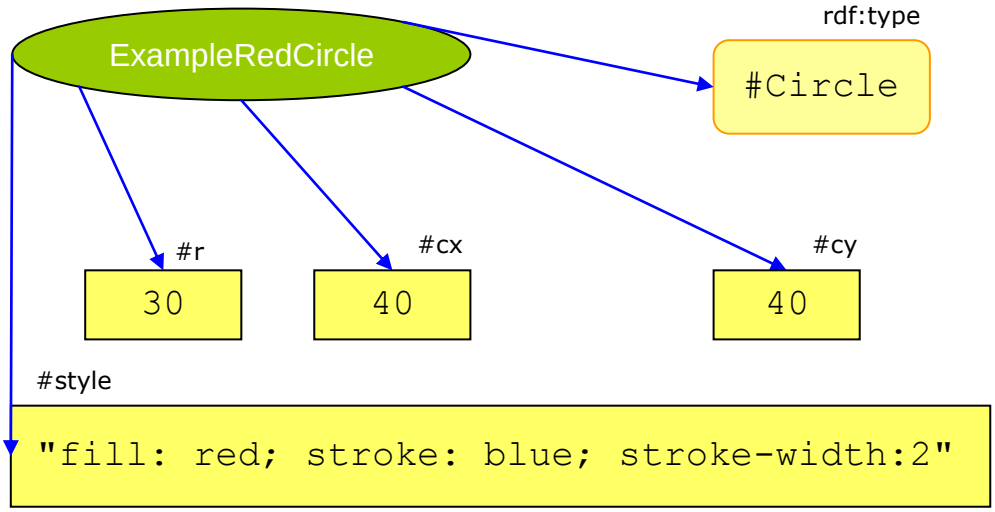
RDF/XML

Resource Description Framework is a metadata modeling language, used to describe relationships between resources. These descriptions are modeled as subject predicate object triples, which are visualized as directed labeled RDF graphs. RDF/XML is an XML based serialization of those RDF triples or graphs. The following is a representation of the same circle in RDF/XML:

```
<rdf:Description rdf:about="ExampleRedCircle">

  <rdf:type
   rdf:resource="#Circle"/>
   <r>30</r>
   <cx>40</cx>
   <cy>40</cy>
   <style>"fill: red; stroke: blue; stroke-width:2"</style>

</rdf:Description>
```



RDF graph showing Subject – Predicate - Object triples

XSLT

Extensible Stylesheet Language Transformations or XSLT, a member of the XSL family of languages, is a Turing complete, functional programming language whose purpose is to transform XML data into other XML languages, text or other data forms.

As a template-based tree navigation and manipulation programming language, XSLT incorporates a vastly different style of programming when compared to common imperative languages, such as C, C++, and Java. For inexperienced users, debugging a program written in XSLT can often prove to be frustrating and time consuming. This is, in part, due to the non-sequential nature of template invocation. When using XSLT for the first time, many programmers try to use it in a more conventional way.

History

XSLT began as a successor to DSSSL (Document Style Semantics and Specification Language), a language used to define transformations for SGML.

There are currently two specifications for XSLT. The first official recommendation (XSLT 1.0) was put forth by the W3C on November 16, 1999. The W3C released a follow-up specification (XSLT 2.0), January 23, 2007; this new recommendation added support for XPath 2.0, multiple output targets, user-defined functions and regular expression matching, while deprecating result tree fragments.

XSLT Basics

Like all other proper XML documents, an XSLT stylesheet should begin with an XML declaration which specifies both the version of XML and the character encoding used in the stylesheet.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Similar to inline XML comments, this declaration is separate from the actual XML tree of which the stylesheet consists. Unlike comments, there may only be one such declaration per stylesheet, and it must appear before the actual root of the document's element tree.

An XSLT stylesheet begins with an element called <xsl:stylesheet> or <xsl:transform>, but this choice is purely aesthetic, as both are treated identically by XSLT processors. Most developers use the xsl:stylesheet as default.

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:stylesheet>
```

This element assumes the role of the root of this particular XML tree, all other instructions, templates and content is defined inside. As the root, the attributes of this element specify the version of XSLT being used (1.0 or 2.0) as well as any namespaces used in the stylesheet. By default, the XSLT namespace is required in all stylesheets, but other namespaces may have to be defined in this element as well.

XSLT Template Matching

A large portion of XSLT's power, and initial difficulty for beginners, comes from its template matching facilities.

The following is the basic structure of an XSLT template.

```
<xsl:template match="/">

<!-- xslt code here -->

</xsl:template>
```

This is the most common template used in any stylesheet, its matching criteria consists only of the root node '/'. The criteria specified in the 'match' attribute for a template can be any valid XPath expression. A single template can match multiple target subtrees with the same element name. If the XPath '|' operator is used, even different element names can be matched.

```
<xsl:template match="Tea | Coffee">
```

XPath is a very robust language, and is used extensively in many XML processing languages. For more information about such expressions, please refer to the section on XPath.

XSLT Control Structures

Iteration

While template matching is the backbone of XSLT, there is also support for iterative style programming. The `<xsl:for-each>` instruction tells the processor to find all nodes that match the specified select criteria, create a node set and iterate through each element of the node set.

```
<xsl:for-each select="{XPATH_EXPRESSION}">
```

Branching

In all programming languages, there must be some way to make decisions based on input. For simple conditions the `<xsl:if>` statement can be used. Such a statement contains a test attribute which evaluates an XPath expression. If the expression evaluates to true, the XSLT code inside the element is applied to the current node.

```
<xsl:if test ="{XPATH EXPRESSION}" >
  <!—XSLT code here -->
</xsl:if>
```

When there are more choices to consider, XSLT has support for classic if-then-else branching. This is accomplished through the use of `<xsl:choose>`.

```
<xsl:choose>
  <xsl:when test="{XPATH EXPRESSION}">
     <!—XSLT code here -->
  </xsl:when>
  <xsl:otherwise>
     <!—XSLT code here -->
  </xsl:otherwise>
</xsl:choose>
```

Variable Binding

As a kind of functional language, XSLT is side-effect free. Processing of the XML tree uses bind-once (single-assignment) variables. Once a variable is bound, it cannot be unbound or changed.

```
<xsl:variable name = "varname" select="{XPATH_EXPRESSION}">
```

After a variable is bound, it can be accessed by using the variable name prefixed by a dollar sign, such as `$varname`.

When calling a template, parameters can be passed from outside the template by means of `<xsl:param>` and `<xsl:with-param>`.

Constructing new XML using XSLT

In the body of a template, new XML elements are created by simply typing the tags we want and annotating them, or we can form elements by them using XSLT's `<xsl:element>` combined with `<xsl:attribute>` in the following manner:

```
<xsl:element name="$elem">
     <xsl:attribute name="$attr">attrValue</xsl:attribute>
      value
     </xsl:element>
```

If $elem is bound to “colour” and $attr is bound to “blue” this will produce the following element:

```
     <colour attr="blue">value</colour>
```

New Features for XSLT 2.0

One of the most prominent new features in XSLT 2.0 is its much increased support for grouping. This allows XSLT to operate on groups of nodes, rather than a single node at a time. Node groups are formed in the following manner:

```
<xsl:for-each-group select="{XPATH EXPRESSION}"
                    group-by="@attribute">
```

In an earlier example, the style information for the circle was represented without markup in a semi-colon delimited list. RuleML/XML has a list representation called a <Plex>, and for users of RuleML, it would be beneficial if this style information was marked up in such a way. This transformation is possible using regular expression matching. For any such list, we can use:

```
<xsl:variable name="tokenList">
  <xsl:value-of select="tokenize($fullList,';')"/>
</xsl:variable>
```

Then we can iterate through the resultant list using:

```
<xsl:for-each select="$tokenList”>
```

Once selected, we apply any formatting necessary to each part of the list.

This new specification for XSLT provides the ability to create user-defined functions, which allows the language to be considered a fully-functional programming language.

```
<xsl:function>
```

Also new in XSLT 2.0 is the ability to output to multiple result documents during execution. This can be accomplished using:

```
<xsl:result-document>
```

XSLT 2.0 incorporates many other new features and changes, which add considerable depth to the language.

XPath

XSLT uses XPath expressions to navigate through XML trees. XPath expressions can be seen as being similar in structure to Unix-like file systems, with extra operators added to allow for matching XML structures, such as attributes, and comparisons, such as greater than or less than. Such operators are known as node axes and node tests, and they allow a great deal of freedom for locating information in the tree.

```
node-axis::test()
```

The most common node axes have shorthand versions, which are more closely in line with the file system-like syntax mentioned above.

```
'.'  self
'..' parent
'//' descendent or self
'@'  attribute
```

XPath also provides many built in functions, which allow string manipulation, boolean evaluation, and node operations. XPath is a subset of XQuery.

XSLT Processing

As a completely interpreted language, XSLT requires a processor which is capable of interpreting and executing instructions in order to process XML documents. Among the many available processors, the two standalone open source processors that are most widely used are Saxonica's Saxon and the Apache project Xalan. Additionally, most modern web browsers and application platforms have some sort of XSLT processor built in to allow for XML translation during runtime.

Saxon

Saxon is an XSLT/XQuery processor developed by Michael Kay which uses SAX as its base. Saxon is available in two versions, a commercial release Saxon-SA (a schema-aware interpreter), and an open source version Saxon-B, released under the Mozilla Public License.

Xalan

Xalan is an XSLT processor developed by the Apache Software Foundation. Originally a project under the direction of IBM, then part of the Apache XML project, it has branched of into a stand-alone project. Xalan currently has two implementations, Xalan-Java which uses JAXP and Xalan-C++, which uses Xerces.

Styles of Using XSLT

As previously mentioned XSLT can be viewed as a non-deterministic template-based programming language, and as such is very different from many languages that average computer programmers will be familiar with. Coming from this difference, there are two main approaches to programming XSLT.

Most programmers are used to imperative language models, such as C, C++, and Java, which use control structures, such as if statements and do – while/for iterations to dictate the flow of the program. These models are available in XSLT to an extent; however XSLT's power lies in its functional characteristics, and many programmers would do best to try to learn this intended way to use XSLT. This involves the use of rule-like templates. Templates work on the basis of matching certain tree patterns. The term 'rule-like' derives from comparison to other functional languages, where language expressions are rules, containing a head and a body. Because of multiple head match possibilities, it may be more difficult for inexperienced programmers to predict and control how they are applied.

That is not to say that when programming in XSLT, a developer must stick to one style or another. Templates may be called at any time using `<xsl:apply-templates>` which will apply all templates that are included in the stylesheet, relative to the currently selected node. This application may be refined to only apply to a certain child or certain children of the current node using the select attribute

```
<xsl:apply-templates/>
<xsl:apply-templates select = "Child1"/>
<xsl:apply-templates select = "C1 | C2 | .. | CM"/>
```

Fixing the subtree to which a template will be applied using select="Child1" may still not be enough control, as you may only want to apply a single, specific template to it. For such situations XSLT provides the following method, analogous to a function call in deterministic functional programming languages:

```
<xsl:call-template name="name-of-template"/>
```

The following examples perform a conversion between RDF/XML and RuleML/XML, as exemplified earlier, and show a marked difference in the usage styles described above. The first example uses only `<xsl:for-each>`, avoiding templates other than the root match. This transformation is performed by first iterating through all elements on the toplevel of the document, a classic nested for-loop. The second example uses only templates. Please note that these translations are only for a small subset of both languages.

Converting from RDF/XML to RuleML/XML using an iterative programming style:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  exclude-result-prefixes="rdf">
<xsl:output method="xml" omit-xml-declaration = "yes"/>

<xsl:template match = "/">
  <xsl:for-each select="//rdf:Description">
    <xsl:variable name = "rdfabout" select="@rdf:about"/>
     <Atom>
       <Rel>object</Rel>
       <slot>
         <Ind>id</Ind>
         <Ind><xsl:attribute name="type">
                <xsl:value-of
                 select="rdf:type/@rdf:resource"/>
              </xsl:attribute>
              <xsl:value-of select="$rdfabout"/>
         </Ind>
       </slot>
   <xsl:for-each select="node()">
     <xsl:if test="not(empty(text()))">

       <xsl:variable name="toplevel" select="."/>
       <xsl:variable name="toplevelName" select='name()'/>

       <slot>
         <xsl:element name ="Ind">
           <xsl:value-of select="$toplevelName"/>
         </xsl:element>
         <xsl:element name ="Ind">
           <xsl:value-of select="$toplevel"/>
         </xsl:element>
       </slot>
      </xsl:if>
    </xsl:for-each>

     </Atom>
  </xsl:for-each>

 </xsl:template>
</xsl:stylesheet>
```

Converting from RDF/XML to RuleML/XML using a template-based programming style:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  exclude-result-prefixes="rdf">
<xsl:output method="xml" omit-xml-declaration = "yes"/>

<xsl:template match = "/">
  <xsl:apply-templates/>
</xsl:template>


<xsl:template name="rdfDesc" match = "//rdf:Description">
  <xsl:variable name = "rdfabout" select="@rdf:about"/>

     <Atom>
       <Rel>object</Rel>
       <slot>
         <Ind>id</Ind>
         <Ind>
           <xsl:attribute name="type">
             <xsl:value-of
               select = "rdf:type/@rdf:resource"/>
           </xsl:attribute>
           <xsl:value-of select = "@rdf:about"/>
         </Ind>
       </slot>

     <xsl:apply-templates/>
     </Atom>
</xsl:template>

<xsl:template name="slot" match =
"//rdf:Description/child::node()[not(empty(text()))]">
  <xsl:variable name = "slotname" select = "name()"/>

     <slot>
       <Ind><xsl:value-of select="name()"/></Ind>
       <Ind><xsl:value-of select="text()"/></Ind>
     </slot>

</xsl:template>

</xsl:stylesheet>
```

Alternatives

There are a variety of other languages and libraries which can serve as alternatives to processing XML with XSLT. These range from simple object modeling libraries to full languages, and are discussed briefly below.

XQuery

The first and foremost competitor to XSLT is the deterministic functional language XQuery. Both languages were developed simultaneously by separate working groups within the W3C, and are similar in almost every aspect. However, the area in which they differ greatly is in purpose. XQuery was designed to allow SQL-like queries to large pools of XML data, whereas XSLT was designed to be document-centric, providing more robust formatting capabilities. XQuery is a significantly smaller language, lacking XSLT's template processing. XQuery should feel very natural for users of SQL. The principal component of XQuery is the FLWOR (For, Let, Where, Order-by, Return) expression.

FOR	Base of query, creates a set of nodes to be operated on and binds them to a variable. For statements can be nested arbitrarily deep. `for $x in //example`
LET	Variable bindings `let $y := //stuff [value = $x]`
WHERE	Restricting conditions on queries, such as Y > 1 `where $y/stuff = $x` Such restrictions can be combined using {and | or} to add precision.
ORDER-BY	Return Results sorted according to the given criteria `order by $x, $y/stuff`
RETURN	Output target, the destination and formatting of the query results `return //thing[@value = $y/reference]`

As XQuery is a superset of XPath, any XPath expression is legal in FLWOR expressions.

The following is an XQuery version of the transformation we performed earlier from RDF/XML to RuleML/XML.

```
declare namespace rdf="http://www.w3.org/1999/02/22-rdf-
syntax-ns#";

declare function local:makeSlot($x){

  for $y in $x/child::node()
    let $slotname := $y/name()
    let $slotfiller := $y/text()
      where not(empty($slotfiller))
        return

          if(not($slotname="rdf:type"))
          then <slot>
                 <Ind>{$slotname}</Ind>
                 <Ind>{$slotfiller}</Ind>
               </slot>
          else ()


     };


     for $object in doc('redCircle.rdf')//rdf:RDF
       let $children := $object/child::node()
       let $descName := $children/@rdf:about
       let $rdfType := $children/rdf:type/@rdf:resource
         return
           <Atom>
             <Rel>object</Rel>
             <slot>
               <Ind>id</Ind>
               <Ind type="{string($rdfType)}">
                         {string($descName)}
               </Ind>
             </slot>
             {local:makeSlot($children)}
           </Atom>
```

XQuery vs. XSLT

In the XML community there is a great deal of discussion concerning these two languages. As XQuery lacks document-centric template processing, it was considerably more difficult to generate the proper formatting required to perform the example transformation shown above. Thankfully, there is support for creating user-defined functions, which can serve almost the same purpose when reuse is necessary. The code that results is arguable more readable than the equivalent XSLT.

In a benchmark (XMark Q8 http://monetdb.cwi.nl/xml/) performed by Michael Kay, the supervising editor of the XSLT 2.0 specification, both languages were compared using a query that performed the exact same transformation in both languages. While XSLT performed better than XQuery for small datasets, XSLT's time complexity was found to be quadratic $O(n^2)$, whereas the same query done in XQuery showed linear complexity, O(n). This quadratic complexity appears to be due to XSLT's search for a matching template for each selected node, or due to a lack of operation specific optimization, due to XSLT's focus on document formatting.

This further exemplifies the difference in design purpose between the two languages, where XQuery is optimized for querying large XML databases, and XSLT is better suited to formatting individual small to midsized documents.

Others

XSLT and XQuery are two robust approaches to working with XML; however, these two technologies may not be suitable for every situation. The following packages are lower level approaches to XML processing; in fact, some of these packages form the basis for most of the XSLT and XQuery processors available.

SAX

The Simple API for XML, this API forms the basis for many XSLT processors, including the popular Saxon and Xalan.

DOM

Document Object Model, is a language independent object model for representing XML (and HTML) documents. A W3C standard, DOM has been widely supported in web browsers for many years now, and is a core requirement for Javascript to function correctly.

Xerces

Xerces is a family of software packages designed for processing XML that is widely used in APIs that support XML processing. There are Xerces implementations in Java, C++, and Perl.

XOM

XML Object Model is an open source API for processing XML developed by Elliotte Rusty Harold. XOM is released under the LGPL.

JDOM

Another open source XML API, JDOM is Java based and combines DOM and SAX, as well as supporting XPath and XSLT.

JAXP

Developed by Sun Microsystems, the Java API for XML processing provides an higher level Java interface for both DOM and SAX, as well as providing and interface to XSLT itself.

Conclusion

XML is becoming increasingly popular and abundant. As such, there is and will continue to be a need to convert knowledge from one XML format to another. XSLT is a language with a lot of potential applications, and many novel features. Due to its tendency towards quadratic complexity, the use of XSLT's template style should be limited to small and medium sized XML databases and to document formatting. On the other hand, XQuery is very efficient when dealing with large datasets and XML databases, but lacks template processing and certain formatting features that are present in XSLT.

Application engineers should consider using XQuery over XSLT for large databases. Should neither of these languages be suitable for the task at hand, there are many other XML processing languages and libraries available to develop suitable solutions for their tasks.

Bibliography

Reading :

Knowledge Markup Techniques
http://www.dfki.uni-kl.de/~boley/kmt/kmt-struc.html

O'Reilly What's new in XSLT 2.0
http://www.xml.com/pub/a/2002/04/10/xslt2.html

The XML Bible, 2nd Edition Rusty Elliot Harold

XML Processing Moves Forward XSLT 2.0 and XQuery 1.0
http://www.xmlprague.cz/2005/slides/kay1.ppt

XQuery: Reinventing the Wheel?
http://www.xmlportfolio.com/xquery.html

Comparing XSLT and XQuery
http://www.idealliance.org/proceedings/xtech05/papers/02-03-01/

Technologies

Extensible Markup Language (XML)
http://www.w3c.org/

XSL Transformations (XSLT)
www.w3.org/TR/xslt

XML Path Language (XPath)
http://www.w3.org/TR/xpath

XML Query Language (XQuery)
www.w3.org/TR/xquery/

Saxon
http://www.saxonica.org
Xalan
http://xalan.apache.org/

Simple API for XML (SAX)
http://www.saxproject.org

Document Object Model (DOM)
http://www.w3.org/DOM/

Xerces
http://xerces.apache.org/

XML Object Model (XOM)
http://www.xom.nu

Java Document Object Model (JDOM)
http://www.jdom.org/

Java API for XML Processing (JAXP)
https://jaxp.dev.java.net/