CS2 Current Technologies note 3

XSL and XQuery

Peter Buneman

3.1 Introduction

We are going to review two methods of transforming XML. The first, XSLT, is dsigned to turn XML into instructions that can be understood by a "rendering" program – a program that is designed to display a document on your screen or on a printed page. Common uses for XSL would be

- to generate HTML,
- to generate LaTeX, or the internal format of some WYSIWYG editor,
- to generate plain text,
- to "internationalise" documents, or
- to account for format conventions

The term "internationalise" does not imply any ability of XSL to do language translation, however one does want to account for different formatting styles (e.g. languages whose display of text is left-to-right or top-to-bottom). Another example would be the internationalisation of a web site. If you look, for example at the UoE home pages, much of the text is formulaic ("search", "contact", "departments", "people", "proceed to checkout" – well maybe not!) The idea is that one could translate these pages by providing translation tables for these terms and generating the pages on the fly.

As for format conventions, consider the presentation of citations such as 'H. Harris, L. Lewis and M. Mull. "How to Comment your Code". *J. Computational Theology* 12(4), 123-345, (2001).' Each subject, in fact each journal, has its own conventions on how to format a citation: Do the authors or the title come first? Are titles italicised? Do initials come before or after the authors' names? Etc. etc. However, the underlying structure (which is easily described in XML) is the same.

For these reasons, XSLT is built into recent versions of browsers such as Netscape and Explorer. XML is downloaded and a page is generated by an XSL style sheet that will take into account your preferences.

3.2 XSLT

At heart, XSL is a simple rule-based language. Rules are of the form "if you see x in the input, generate y in the output." Here is a simple example. We have our employees and projects example, repeated again in Figure 1 and we want to produce a list of the names of the projects together with the names of the employees working on those projects. Our goald is to produce some HTML which will result in the following display:

- Pattern recognition: Mary Joe
- XML compression: Mary Jane

Note that the structure of the output follows the structure of the document, and this is what style-sheets are designed to generate.

Here is our first attempt:

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

<xsl:template match="/">
<html>
<ul>
</ul>
</html>
</xsl:template>

<xsl:template match="project">
<li>
    <xsl:value-of select="name"/>
</li>
</xsl:template>

</xsl:stylesheet>
```

First of all, note that the XSL program is also well-formed XML. The "`xsl: ...` " prefix on the tags says that these tags are within the scope of a *namespace* – a simple idea which we will discuss later. The fact that the syntax of this simple program is not particularly readable already causes us to doubt that XML is a good syntax for all forms of structured data[1].

The program contains two rules which are triggered when a node is matched with `<xsl:template match="...">`. The match patterns, `"/"` and `"project"` are specified in XPath and tell us what to do on encountering nodes that match these patterns. In the case of the root of the document, we generate the "top level" HTML. In the case of a `"project "` node we generate the list elements. We also insert the "value" of a node, selected by another XPath expression, `"name"`

Unfortunately, the program as it stands doesn't work properly. All it generates is: `</html> <ul> </ul> </html>`. The problem is that as soon as a match is encountered, the node is

[1]You should be aware that one of the very first programming languages, Lisp, also uses a "labelled brackets" notation. The idea clearly did not catch on as a user-friendly syntax for programs!

```
<db>
    <project>
      <name> Pattern recognition </name>  <budget> 10000 </budget>
      <members>
        <employee>
          <name> Mary  </name>  <age> 34  </age> <allocated> 25%  </allocated>
        </employee>
        <employee>
          <name> Joe  </name> <age> 36 </age> <allocated> 50% </allocated>
        </employee>
      </members>
    </project>
    <project>
      <name> XML compression </name> <budget> 90000 </budget>
      <members>
        <employee>
          <name> Mary </name> <age> 34 </age> <allocated> 50%  </allocated>
        </employee>
        <employee>
          <name> Jane </name> <age> 22  </age> <allocated> 25% </allocated>
        </employee>
      </members>
    </project>
</db>
```

Figure 1: The Projects XML file

treated as "processed", i.e., no further processing takes place on the subtree of that node. In order to continue to apply templates to the subtree of the node, we add an explicit injunction, `<xsl:apply-templates/>` to continue the matches, now using this node as the *context* node. Thus we change the "root" rule to:

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

This gives us a list of the project names. Now let's try to add the employee names after each project. Presumably we repeat the idea by changing the rule for `project` to continue processing subnodes and adding a rule for `employee`:

```
...

<xsl:template match="project">
<li>
    <xsl:value-of select="name"/>
    <xsl:apply-templates />
</li>
```

```
</xsl:template>

<xsl:template match="employee">
    <xsl:value-of select="name"/>
</xsl:template>
```

This almost works, but there it outputs a lot of stuff we didn't want. The reason is that the `xsl:apply-templates` in the `match="project"` rule caused the XSL processor to look at the submodes of `project` nodes, and for some of these nodes there is now no matching rule. So the "default" rule for those nodes is invoked, which is simply to include the text under that node.

To prevent this happening, we change the selection in the `xsl:apply-templates` rule to `<xsl:apply-templates select="members" />`. Other child nodes of `project` nodes are then ignored by the processor.

Our program now looks like this:

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

<xsl:template match="project">
<li>
    <xsl:value-of select="name"/>
    <xsl:text>:</xsl:text>
    <xsl:apply-templates select="members"  />
</li>
</xsl:template>

<xsl:template match="employee">
    <xsl:value-of select="name"/>
</xsl:template>

</xsl:stylesheet>
```

To within some annoying whitespace (there must be a function for getting rid of this) we get the desired output. The `xsl:text` is for including text in the output.

It's important to understand how pattern matching works. For example, if we change the `value-of` in the `project rule` to `<xsl:value-of select="//name"/>`, we get the following output:

- Pattern recognition : Mary Joe
- Pattern recognition : Mary Jane

Why does this happen? Also, with the machinery you have already seen, it is easy to make XSLT go into an infinite loop. How? In our examples so far we have only used the XPath child and descendant

axes. The other axes can also be useful. If we want to do a better job of punctuation, we could replace the `employee` rule by the following two rules:

```
<xsl:template match="employee[following-sibling::*]">
    <xsl:value-of select="name"/>
    <xsl:text>,</xsl:text>
</xsl:template>

<xsl:template match="employee[not(following-sibling::*)]">
    <xsl:value-of select="name"/>
    <xsl:text>.</xsl:text>
</xsl:template>
```

XSLT has much more than this. There are iterators, conditions and other constructs which make the language look less like a pattern matching language and more like a programming language. But this comes at a price: less control over the efficiency of the programs, less hope of typechecking, and almost no hope of using these languages on very large data sets. Rather than investigating these additional features of XSLT, we shall look at a language for which there is some such hope, XQuery.

3.3 XQuery

XQuery is the latest, and best established, in a succession of attempts to build a proper query language for XML – something that might do for XML what SQL does for relational databases. XQuery can be seen either as a "glue" that turns XPath node sets back into XML, or it can be seen as variant on SQL (which in some sense it is) for XML.

The core of XML is a variant of *list comprehensions* that are used in functional programming languages and Python, and which can also be used as a model for SQL and other database query languages.

The simplest form of an XQuery query is:

for $\$x$ in p
where c
return e

Here p is an XPath expression, c is a condition, and e is an XML expression, which may include variables and other XPath or XQuery expressions.

For example, to obtain the names of the projects with budgets in excess of 5000 one could use the query:

```
for    $x in document("projects.xml")//projects
where  $x/budget > 5000
return $x/name
```

The result is to produce `<name> Pattern recognition </name> <name> XML compression </name>` ... . Note that this is not yet well-formed XML. To make it so we would include the query in an enclosing tag such as `<answer>{ ... }</answer>`.

To get the names of projects to which Jane is assigned:

```
for    $x in document("projects.xml")//projects
where  $x//employee/name ="Jane"
return $x/name
```

Remember note that `$x//employee/name` will typically yield a set of names. Remember the rules governing equality in XPath? They apply in XQuery too. Here is a query that shows nesting of queries within XML. What will the output look like?

```
<answer>{
   for  $x in document("projects.xml")//projects
   return
     <project>{
        $x/name,
        <people>{
          for $y in $x//employee
          return $y/name
        }</people>
     \}</project>
}<\answer>
```

It is important to distinguish between sets of nodes and sets of values. The query

```
   for  $x in document("projects.xml")//employee/name
   return $x
```

will return the `<name> Mary </name> <name> Joe </name> <name> Mary </name><name> Jane </name> ...` . A set of *nodes* in the order in which they occur in the document. Sometimes, we want to get the distinct values. For example:

```
<db1>
   for $x in document("projects.xml")//employee/name
   return
    <person>{
       $x,
       <projects>{
          for $y in document("projects.xml")//projects
          where $y//employee/name = $x
          return <projname> $y/name </projname>
       }</projects>
    }</person>
</db1>
```

This shows an example of *restructuring* a document. This is something that is common in database queries. This can also be done in XSLT, essentially by embedding a program that looks a bit like this into XSLT.

The following are taken from the XML "use cases" document – designed to show off what XQuery can do. They illustrate the use of aggregate functions and the use of `let` to bind a new variable to a value or set of values.

List each publisher and the average price of their books.

```
for $p in distinct(document("bib.xml")//publisher)
let $a := avg(document("bib.xml")//book[publisher = $p]/price)
return
  ⟨publisher⟩
    ⟨name⟩{$p/text()}⟨/name⟩
    ⟨avgprice⟩{$a}⟨/avgprice⟩
  ⟨/publisher⟩
```

List the publishers who have published more than 100 books.

```
⟨big-publishers⟩
  {
    for $p in distinct(document("bib.xml")//publisher)
    let $b := document("bib.xml")//book[publisher = $p]
    where count($b) > 100
    return $p
  }
⟨/big-publishers⟩
```

3.4 Some URLs

Note: documents on XQuery typically describe XPath too.

- Nice tutorials on both XPath and XSLT through examples: `http://www.zvon.org/xxl/`
- The XQuery specification (impenetrable): `http://www.w3.org/TR/xquery`
- XML Query Use Cases. A set of examples used to "show off" XQuery (or maybe to test implementations). Lots of examples. Much more readable than the standard/ `http://www.w3.org/TR/xmlquery-use-cases`
- A nice, straightforward, tutorial: `http://www.brics.dk/~amoeller/XML/querying/`
- An interesting paper showing how XQuery can be typed. Quite readable even if you are not interested in types! `http://homepages.inf.ed.ac.uk/wadler/papers/...`
  `... xquery-tutorial/xquery-tutorial.pdf`