CS2 Current Technologies note 2

Programming with XML. XPath

Peter Buneman

2.1 Programming with XML

As we saw in the first set of notes, the idea behind XML is that it should be both human- and machine-readable. Because of the layers of complexity that are being added to the underlying simple idea of XML, people are having increasing reservations about the degree to which XML is easily readable by humans; but if we avoid the clutter it is a reasonable presentation for data structured in a labelled tree. Moreover, it is certainly the case that XML is here to stay as a standard format for *data transmission*, so we need to understand it whether or not we want to read it.

What about machine readability? The whole point of having a standard is that at least it removes from us the low-level parsing problems. If you look at the code for applications that interpret, say, configuration files, a substantial amount of the code is concerned purely with parsing. Now parsing is a "solved" problem in computer science. There are parser generators like YACC (Yet Another Compiler Compiler) which allow you to specify a parser more or less declaratively. But YACC is still quite a complicated tool and is much more than is needed for a supposedly "parsing-friendly" format. What we want are tools or languages that present us with parsed XML. Here we shall discuss in varying degrees of detail, these tools and languages. We shall cover:

SAX (Simple API for XML) where API stands for Application Program Interface. This is a low-level interface which does little more than "traverse" the document. You would use SAX from a "host" language such as Java, C++ or Python.

DOM for document object model. Again, this is an API, but here the nodes of the document appear as objects in the programming language, and there are methods, for example, for obtaining the child nodes of a node and for traversing the document in various ways.

XSLT XSL stands for Extensible Stylesheet Language. The "T" has been tacked on for "transformation", to make up for some deficiencies in the early versions of this language. The idea of a style sheet is to turn XML into some language for the presentation/display of text such as HTML. Of course, there are many such presentation languages. Roughly speaking, XSLT is useful when the structure of the presentation follows the structure of the XML. So, considering the example in Figure 1, it is relatively easy to write an XSLT program that produces the HTML that will construct a tabular output such as

Projects

Name	Budget	Employees
Pattern Recognition	10000	Mary Joe
XML Compression	90000	...
...	...	...

However it is considerably harder to produce a table of employees with their names and ages and the departments they work on. For this we need to turn the structure of Figure 1 "upside down". While this is possible in XSLT, it may be quite clumsy.

XQuery This is the latest in a succession of query languages proposed for XML. The idea is that it can express XML-to-XML transformations such as "inverting" Figure 1. Another, as yet unrealised, hope for these query languages is that we shall be able to make them as efficient as database query languages such as SQL. This involves not only understanding how physically to store the XML (storing it as text won't work) but also understanding how to capitalise on the *type* of XML.

The last two languages, XLST and XQuery, are dependent on a language that selects "nodes of interest" from a document, so in order to understand these languages, we need to understand XPath. But first let's look briefly at the APIs.

2.2 SAX

The idea behind SAX is very simple. SAX makes a single traversal of the document. As it encounters tags etc. it calls procedures which *you* write – by implementing a "virtual" method. Here is a sample of the more important procedures (I doubt that the names are the ones I've given here)

- `procedure open(tag)`
  What to do when an opening tag is encountered. The parameter `tag` is the name of the tag.
- `procedure close(tag)`
  What to do when an opening tag is encountered.
- `procedure charinput(text)`
  What to do when PCDATA is read in (and bound to the parameter `text`).

Using these, suppose we are asked to print the names of employees frm the XML in Figure 1.

The pseudocode is as follows:

```
procedure init (){    // called at the start of the scan
  in_employee := false;
  in_name := false;
}
```

```
⟨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⟩
       ...
      ⟨/members⟩
    ⟨/project⟩
    ...
  ⟨/db⟩
```

Figure 1: Yet another representation of employees and projects

```
procedure open(tag) {
   if tag = "employee" then in_employee := true
   if tag = "name" then in_name := true
}

procedure close(tag) {
   if tag = "employee" then in_employee := false
   if tag = "name" then in_name := false
}

procedure charinput(text) {
   if in_employee and in_name then print(text);
}
```

This is all pretty simple and obvious, but now suppose you are asked to print, for each employee, the name followed by the age. Are we guaranteed that the name precedes the age in the source document? If not we need to do some buffering. To make matters worse, what do you do if an employee has more than one name and more than one age? There's no guarantee that names and ages exist or are unique. You rapidly find that programs get more complicated and may involve multiple passes through the document or building auxilliary data structures such as index tables.

To summarise: SAX is simple and efficient. Moreover there is no limitation (imposed by SAX) on the size of the document. Of course, your program may need to create all sorts of intermediate data structures if a simple scan is not appropriate to the query.

2.3 DOM

In the document object model, nodes are objects with various methods. For example, one can ask for the tag of a node and one can ask for the ith child of a node – essentially the children form an array of nodes.Our pseudo-code for DOM (again the names of the procedures are fictitious.) is as follows:

```
   root := create_root("projectfile.xml");
   for n in root.children() {
     for m in n.children {
         if m.tag= "members" {
           for l in m.children {
             for k in l.children() {
               if k.tag = "name" {
                 print k.text_contents()
               }}}}}}
```

Note that we have assumed a lot about the structure of the document for this to be the correct answer. The DOM has some additional procedures such as one to find all the *descendant* nodes

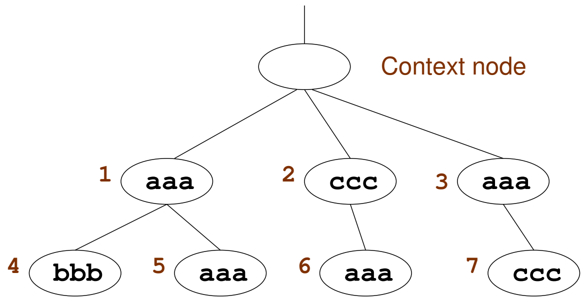

Figure 2: A tree for XPath examples

with a given tag. Using this, the procedure above can be shortened and can be made more robust in the sense that it would work on a wider variety of representations.

To summarise, the DOM is closer to conventional programming with main-memory data structures. Complex programs are typically easier to write in DOM than SAX. On the other hand, there are only a few DOM implementations that will work on large bodies of XML, and on large bodies of data, one would like to take advantage of optimisations such as those performed in relational query languages. However, DOM is not sufficiently "declarative" for such optimisations to be easily extracted from the code.

2.4 XPath

Xpath itself is neither an API nor a query language. It is a method for describing a set of nodes in an XML document. As such it is used both in XSLT and XQuery, so studying it is a necessary precursor to understanding these languages. Our SAX and DOM examples already hint at the need for a language for specifying a set of nodes. In various ways we tried to find the `name` name nodes that were somewhere below an `employee` node. In XPath this set is simply specified as `//employee//name`. What this specifies is a set of paths from the root of the document each of which contains an `employee` tag and ends in a `name` tag.

Like everything associated with XML, the core idea is very simple and based on well-understood ideas. However there are numerous and arguably unnecessary "bells and whistles" that make understanding XPath quite tricky.

In XPath we think of ourselves sitting at a "context node". XPath specifies a set of paths starting at this node. The nodes we want are the nodes at the end of this path. As a simple example, suppose we are "at" the first `members` node in the XML in Figure 1. The path `employee` is a one-edge path which yields the two nodes below that node. The path `employee/name` gives a set of two-edge paths which end up at a name node.

For some more examples, consider the tree in Figure 2 in which we have given the nodes artifical identifiers.

In its simple forms XPath is similar to "navigating" a unix-style directory:

`aaa`	all the child nodes of the context node labeled aaa {1,3}
`aaa/bbb`	all the `bbb` children of aaa children of the context node {4}
`*/aaa`	all the aaa children of *any* child of the context node {5,6}.
`.`	the context node
`/`	the root node

Some further examples:

`/doc`	all the doc children of the root
`./aaa`	all the aaa children of the context node (equivalent to aaa)
`text()`	all the text children of the context node
`node()`	all the children of the context node (includes text and attribute nodes)
`..`	parent of the context node
`.//`	the context node and all its descendants
`//`	the root node and all its descendants
`//para`	all the para nodes in the document
`//text()`	all the text nodes in the document
`@font`	the font attribute node of the context node

Predicates can be applied to paths. They select a subset of the paths chosen by the XPath expression to the left of the predicate.

`*[2]`	the second child node of the context node. (`*` matches any label)
`chapter[5]`	the fifth chapter child of the context node
`*[last()]`	the last child node of the context node
`person[tel="12345"]`	the `person` children of the context node that have one or more `tel` children whose string-value is `"1234"` (the string-value is the concatenation of all the text on descendant text nodes)
`person[.//firstname = "Joe"]`	the person children of the context node whose descendants include firstname element with string-value `"Joe"`

From the XPath specification (`$x` is a variable – see later):

> NOTE: If `$x` is bound to a node set then `$x = "foo"` does not mean the same as `not($x != "foo")` .

The XPath we have seen so far is an abbreviation for full XPath in which each step can take place in a given direction or along a given “axis”. For example, a step may move to a previous or next sibling. In most cases we will only move along the “downward” axes, but it’s important to note that the XPath specification allows a variety of axes.

`ancestor::employee`: all the `employee` nodes *directly above* the context node `following-sibling::age`: all the age nodes that are *siblings* of the context node and to the *right* of it.

`following-sibling::employee/descendant::age`: all the age nodes *somewhere below* any `employee` node that is a *sibling* of the context node and to the *right* of it.

`/descendant::name/ancestor::employee`:
Same as `//name/ancestor::employee` or `//employee[boolean(.//name)]`