Extracting data from XML

Michael R. Levy

Dept. of Computer Science, University of Victoria
P.O. Box 3055, Victoria BC, Canada V8W 3P6
(email: mlevy@uvic.ca)

SUMMARY

We propose a system that can be used by client-server systems to produce output from XML documents. This system is an alternative to both the direct programmers API approach, and the XSL rendering approach.

KEY WORDS: XML, Template Resolution, TML, Guide, CGI

INTRODUCTION

XML\(^1\) is fascinating for being boring. It is a simplified version of SGML, and offers nothing much beyond a grammar standard for describing structured documents. Except for hypertext linking, there are no semantics in an XML specification. In spite of this, XML is likely to become a dominant mechanism for the delivery of information over the Internet. We can think of XML as doing for data what HTML did for word-processing: it will establish a niche. This niche will not be adequate for all purposes, but it will be adequate for most “light-weight” purposes. By being widely adopted, all serious players in the document or data delivery game will have to support XML. We are likely to see a growing number of applications that deliver XML documents. These might include conventional databases, monitoring devices, financial services and so on. The problem we address in this paper is the following: What is a good mechanism for translating XML documents? By “good” we mean (at least) elegant, concise, complete, implantable and flexible. This implies a different approach than generic flexible rendering systems such as XSL\(^2\), because (a) we may not wish to render the XML file and (b) we may use use a conventional rendering system, such as an HTML browser or a Postscript processor.

This paper is preliminary work.

PROCESSING XML DOCUMENTS

To focus the discussion somewhat, suppose that we have posed an SQL query to a database via a Web browser, and received back a solution in the form of an XML document. We will further suppose that we do not care about the specific XML language, but that the document is at least conformant with the appropriate DTD. Let us also suppose that the query has one or more answers, each of which has some structure. There are a number of ways such a response might be presented to the user.

1. One solution at a time, with “previous” and “next” links on the page.
2. Ten (say) solutions at a time, with “previous” and “next” links.
3. All the solutions on a single page.

Of course the links will only exist if they should. So the first page will not have a “previous” link, and the final page will not have a “next” link. The individual answers might be displays as paragraphs, lists, tables or using any valid and appropriate HTML tags. A further difficult problem is that we might wish to limit the number of answers displayed based on the amount of space taken by the answers.
How might we solve this problem? The “conventional” way is to add an XML capability to the browsers scripting system. Microsoft already distribute a COM XML object with Internet Explorer version 4\(^3\). With sufficient “scripting”, a programmer can present the XML result in any of the three forms presented above. On possible problem is that the XML object requires the entire XML result to be downloaded to the client before any processing takes place. This is often desirable, but not always. For example, WWW search queries often return a huge number of hits, but are ordered by relevance, so that a user is most likely to look only at page 1 of the response. Waiting for all responses to download is not reasonable. Secondly, there may be security reasons for limiting the particular output that is sent to the browser. Finally, the actual scripting program can be quite cumbersome: the Javascript routines must traverse the XML object, extract tags and attributes, and generate all necessary HTML. By the nature of Javascript, it is difficult to develop helpful general purpose libraries that can be used in all the scripts that need them. Other approaches include the use of XML-aware browsers\(^4\) and the use of XSL. These systems have similar limitations.

In this paper, we propose a Server-side solution to the problem. That is, we assume that the XML document is obtained by the server, and that the translation also occurs on the Server, which then sends the output to the browser. We do not claim that Server-side processing is superior to client-side processing; both have their place. However, we do claim certain advantages of our solution, including:

1. We use Templates (TML\(^5\)). This means that Web designers can use WYSIWYG or other HTML creation tools: they do not have to program all HTML output.
2. Our system is natural and concise: we abstract out those parts of the translation that will be repeated for every XML translation application.
3. Our proposed solution can be used for other XML translations, including XML-XML translation.

**CONTEXT, GUIDE AND TML**

Our solution is based on the mechanism used by Guide\(^6\) for server-side processing. Guide scripts use a “context” which is function from variables to values. We will denote a particular binding as \(x:v\), where \(x\) is the variable and \(v\) is a value, and a context as \(<x_1:v_1,x_2:v_2,...,x_n:v_n>\). Guide scripts always execute in a predefined context. This context is cleverly chosen by the Guide system, so that context control is almost always implicit in Guide scripts. Guide programmers can explicitly create a new context or move to an existing context, using the context: phrase. If the Guide script was launched by an HTML form, the form variables and their values are part of the context. These variables even survive context switching, unless they are explicitly forgotten (using, of course, the forget: phrase). Guide also automatically supports persistence, by saving the current context when a Guide script terminates.

Guide output is produced (usually) by a process known as resolving. A TML file (HTML with substitution tags) is passed to the resolver, which (a) adds appropriate values to form tags; (b) replaces variables inside tags by their values; and (c) substitutes TML tags of the form \(< subst href=x >i\) by \(v\), provided that \(x\) exists within the current context. All of this work is done by the http: phrase. Here, for example, is a typical, useful, Guide script:

```
mailto:mlevy@uvic.ca:response, http:thanks.html;
```

This script uses two TML files: response, which is formatted as a valid mail file, and thanks.html.

An important property that Guide supports automatically is persistence. When a Guide script exits, the current context is saved. If the resolved template is itself a form, the forms associated script will resume within the same context. Automation of this mechanism saves the programmer from the tedium of explicit serialization, saving and loading of context files.

To support database access, Guide has been augmented by an sql: phrase. At present, this phrase automatically extends the current context (if the phrase succeeds). However, to obtain more flexibility, we are planning to modify the sql query so that it produces an XML document. We will use regular tree expressions and pattern matching to extract data from the document, and deliver it to the client.
REGULAR TREE EXPRESSIONS

Regular tree expressions are used to extract data from XML files. Firstly, we note that XML files are trees in the sense that entities do not overlap in well-formatted XML documents. What kind of trees are they? Recall that trees can be defined as nodes with sets of children (un-ordered trees) and nodes with sequences of children. XML trees are hybrids: in some cases, the children must be ordered, while in other cases they must not be ordered. Consider these two examples:

```xml
<p>This <b>must</b> be ordered</p>
```

whose tree is

```
This <b>be ordered</b>
```

and

```xml
<query_result>
  <book>
    <author>Fred</author>
    <price currency="US">10.23</price><qty>6</qty>
  </book>
  <book>
    <price currency="US">12.84</price>
    <author>Tom</author>
  </book>
</query_result>
```

whose tree is

```
<query_result>
  <book>
    <author>Fred</author>
    <price currency="US">10.23</price><qty>6</qty>
  </book>
  <book>
    <price currency="US">12.84</price>
    <author>Tom</author>
  </book>
</query_result>
```

We define now define regular tree expressions. We use variables to extract data when the expression matches a particular tree. A variable is an identifier whose first character is "$". A functor is a term of the form <$t A$s> where $t$ is an identifier or variable, and $A$s is a (possible empty) set of attributes (which may include variables).

A regular tree expression is then defined as follows:
1. <empty> is a regular tree expression.
2. A quoted string is a regular tree expression.
3. A variable is a regular tree expression.
4. If E is a regular tree expression, and $V$ is a variable, then ...$V$ is a regular tree expressions.
5. If $E_1$, ..., $E_n$ are regular tree expressions, then so is $F(E_1, ..., E_n)$ and $F(E_1, ..., E_n)$, where $F$ is a functor.
6. If $E$ is a regular tree expression, so is $^\wedge E$.
7. If $E$ is a regular tree expression, so is $E$?

Intuition

Regular tree expressions can be used as search patterns against XML documents. Consider, for example, the tree shown in the above diagram. Suppose we wish to find all prices mentioned in the tree, regardless of currency. The pattern would be

```xml
<price currency="$c">($p)
```

This pattern matches twice in the above example: the first time yielding ("$c:"US",$p:10.23") and the second time yielding ("$c:"US",$p:12.84").

If we wish to match unordered attributes, we use {} rather than (): for example,

```xml
<book>{{<author>($a),<price $>_($p),...$}_}} will match twice. ? is used for optional matches. The special expression ...$x$, for any variable $x$, matches zero or more children. The variable $$_$ is used for "don’t care" bindings - each occurrence is a different anonymous variable. Finally, the $^\wedge$ symbol is used to denote the root of the tree. We could, for example, force a rooted search for <book> as follows:

```xml
~<query_result>(...$_,<book>{<author>($a),<price ...$as>($p),<qty>($q)}(?)
```

THE MATCHING ALGORITHM

Matching regular tree expressions is a task that is very similar to matching in functional programming languages (like ML), and is also similar to (but simpler than) Prolog’s unification algorithm. The naïve algorithm should work something like this:

1. Build a tree representation of the XML document (a relatively straight-forward task because of explicit design requirements of XML).
2. Build a tree representation of the pattern (trivial).
3. If the pattern is rooted, the matching is straight-forward tree matching.
4. Otherwise, attempt to match the pattern against every subtree, using a depth-first, left-to-right search.

If we call the function implemented by this algorithm $M$, then we say that $M(p,t) = c$, provided that $M(p,t)$ succeeds, and where $t$ is an XML tree and $p$ is a regular tree expression, henceforth known as a pattern.

Of course, the algorithm must work quickly, because data extraction will usually be performed with a human user is waiting at a client somewhere to the see the answer. Some of the techniques used in functional programming languages can be applied to regular tree expressions. Most notably, it most certainly makes sense to compile the regular tree expressions. The basic idea is this: given a regular tree expression $p$, such that $M(p,t) = c$, for any XML tree $t$, the compiler must produce a function $f_p$ such that $f_p(t) = c$.

Unlike Prolog or functional languages, we need to extend the basic algorithm in two ways. Firstly, we need to control the number of matches returned. Secondly, we need to be able to skip matches. We thus add two additional parameters, giving
$M(j, n, p, t)$

where $j$ is the maximum number of matches to be returned, $n$ is a continuation (number of matches to skip), $p$ is a pattern and $t$ is the XML tree. We use -1 as the second parameter to indicate all matches.

An implementation should cache its position in the tree so that it can rapidly skip matches in the usual case that the caller requests $b$ matches at a time, until there are no more matches.

PUTTING IT ALL TOGETHER

The pieces we have are these:

1. A query mechanism $Q$ that takes a query, and whose output is an XML document.
2. A matching algorithm $M$ that takes a pattern, a document and additional parameters, and returns a context.
3. A template resolution function $R$ that takes a template and a context, and produces a resolved document (possibly an HTML document).

That is, given a query $q$, we send the client the result of:

$$R(t, M(1,-1,p,Q(q)))$$

where $t$ is our pretty TML template and $p$ is the pattern.

CONCLUSIONS AND FURTHER WORK

There is much further work to do. If one wishes to use these ideas with Java servelets, it is necessary to implement the resolve and match methods. The compiler would, of course, produce a Java method that would conform to the appropriate interface. Template resolution is quite straightforward. A representation for contexts would have to be chosen. This is built-in in Guide. It ought to be possible to distinguish between single matches and multiple matches in some straightforward manner. In Guide this is done by using arrays.

$R$ has been implemented both within Guide and as a stand-alone Perl script. I am currently working on the implementation of a non-compiled version of $M$.

Regular tree expressions (patterns) provide a concise and natural way to extract data from XML documents. Using a matching method $M$, one can create concise and generic systems that extract and return parts of the document. Combined with Template Resolution, this provides a mechanism for producing responses to queries with minimal programming.

REFERENCES

