xmltr

XML TRANSLATION SUITE FOR FRONTIER

The Xmltr Suite

Xmltr is a suite for UserLand Frontier. You can use xmltr to translate documents marked up with Extensible Markup Language (xml) into alternate representations — such as HTML for publication on the web via the Frontier Web Site Framework or to a tool-specific markup language for print publication.

Table of Contents

What's New?

Version 1.1 of xmltr was released on 24 January 1999. See the Revision History for a list of changes.

A set of proposals for changes to xmltr was posted on 24 November 1999.

Documents in PDF Format

The text of this web site is also provided in pdf format for easy off-line reading and reference. You can download it separately from the sidebar button “Docs in PDF” or as part of xmltr suite archive from the “QuickStart” page.

The pdf document was produced from the same xml source file which generated this web site. It was translated using xmltr into a markup for a print-formatting system (PageMaker), printed to PostScript and then distilled into pdf.

About the Author

Paul Howson is a design and publishing specialist with over 20 years experience in the industry, ranging from traditional typesetting and manual production systems through to electronic and internet publishing. He has also worked professionally as a programmer for a number of years.

Contact Details

Paul Howson
The Design Group Qld
P.O. Box 379
Warwick Qld Australia
Tel +61 7 4661 7720
Fax +61 7 4661 7740
email: phowson@flexi.net.au
xmltr web site: http://www.flexi.net.au/~tdg/xmltr

About Xmltr

Xmltr works together with the blox xml parser from *Technology Solutions* to translate xml into other formats. Blox converts an xml source document into a “parse tree” inside the Frontier object database (ODB). Xmltr traverses the resulting parse tree and invokes a series of **translation rules** corresponding to each markup construct in the source document.

An Overview of Xmltr

Translation Rules

Translation rules can be defined in a number of ways — ranging from simple text substitutions to complex scripts which can perform arbitrary processing. For example, you might use a simple text substitution rule to translate markup tags denoting emphasized text into an HTML <b></b> tag pair. A script translation rule on the other hand might be triggered by a <page></page> tag pair to set up a web page object (complete with directives derived from xml tag attributes) within Frontier’s web site framework (WSF).

Rules are Context Dependent

Translation rules are triggered by the **context** of tags within the xml source document. For example, an <emphasis> tag inside a <par> tag might trigger a different translation rule from an <emphasis> tag appearing inside a <heading> tag. (In the first case, <emphasis> might translate to “bold formatting”, whereas within a heading which is already in “bold format”, <emphasis> might mean “add italic formatting as well”.) If you want to ignore tag context, you can define wildcard translation rules which apply irrespective of the context of the xml tag.

A Close Fit with the Frontier Environment

Xmltr leverages the strengths of Frontier by defining translation rules within tables in the Frontier ODB rather than via an external translation language. Translation rules thus have access to the full power of the Frontier scripting environment and can integrate closely with the xml parse tree, the Frontier Web Site Framework (WSF) and other Frontier suites and interfaces to external programs.

Potential Applications and Benefits of Using Xmltr

Perhaps the most obvious use of xmltr is as an adjunct to the Frontier Web Site Framework. You can author the content of your web site as text marked up with xml tags then translate your source document/s into one or more web pages which can then be published via the WSF.

The author believes this approach can have the following benefits over creating a web site directly in HTML or via Frontier "macros" within the Frontier ODB:

- You can define xml tags to capture arbitrary kinds of structure within your source documents and then translate this structure into HTML. Authoring directly in HTML on the other hand limits you to the pre-defined HTML structural elements.
- Translations can depend upon the **context** of tags within other tags in a document hence minimising the number of different markup tags required.
- By tagging your document's abstract structure, you open the possibility of translation into forms other than HTML. For example you might choose to also translate into QuarkXpress markup tags to produce a good-looking formatted pdf file for users to download and print.
- You can author your web site (or pieces of it) as a text file using your favourite word processor or text editor. When xml structured-editors become readily available, editing these documents will become even easier.
- Your investment in content will be less dependent on (or locked into) a particular publishing tool. You have the option of moving your source (with minimal modification) to other xml-based publishing tools in the future.

Numerous other possibilities also present themselves. For example, there is no reason why remote authoring via Frontier (i.e. where content can be "checked out" and edited via a browser interface) could not be done using xml markup, with blox and xmltr inserted into the web site publishing chain.

Xmltr can also be used as part of a print publishing workflow system. Documents can be authored as text files with xml markup defining the structure and then translated using xmltr into a tool-specific markup language (e.g. LaTex or PageMaker, QuarkXpress or FrameMaker tags). The pdf version of this documentation was translated from the same xml source as the web site and formatted in PageMaker.

Why I Wrote Xmltr

Before taking on web site publishing, I'd been involved in print publishing for over twenty years and was familiar with markup based typesetting systems of various kinds.

It was clear from trying a number of the leading Mac and Windows-based web site authoring tools that the art of document assembly had taken a giant step backwards.

The notion of separating content from format, which had been understood for years in the print world, had been forgotten. The new web authoring tools required tedious manual hard-coding of formatting and construction of “pages” (whether through manual editing of HTML or via a wysiwyg interface).

Neither did these systems account for the need to publish content in a variety of formats — for example web pages **and** printed form (e.g. pdf).

When XML was announced, it seemed the obvious tool for marking up content according to its **structure**. What was needed then was a simple translator from XML to publication markup. The arrival of the blox XML parser made such a tool possible within the Frontier environment.

This is the purpose of xmltr, which is described in the following pages.

The Frontier/blox/xmltr combination is a very effective, lightweight and flexible system for applying generic markup as part of a web or print publishing workflow system.

It leverages the strengths of the Frontier environment by using nested table structures to define translation rules and by allowing translation rules to be arbitrary Frontier scripts.

What About XSL?

I’m aware of so-called “standard” translation languages such as DSSSL and the proposed XSL which is being promoted as “the way” to translate XML. However I believe both these systems are unnecessarily complicated and a much simpler solution such as the one offered here will suffice for many purposes. In addition, the design of XSL has yet to be finalised (as of January 1999).

You can read more of my thoughts on XSL in the section“More on XSL”.

Cautions When Using Xmltr

Unless you have an xml editing tool which ensures files have correct xml syntax, you will need to exercise some caution in preparing xml source files.

The blox parser is (as of January 1999) a non-validating parser — it does not use a Document Type Definition (DTD) to define the allowable structure of the source document. Nevertheless, blox will catch xml syntax errors and unbalanced tags.

It is up to you to invent a suitable schema of xml tags to capture document structure and to apply these in a consistent way to ensure that tags balance and that the document syntax is valid xml.

It is up to you to anticipate the tag patterns which will occur in your document and to write corresponding translation rules. Xmltr will raise a ScriptError (and terminate the translation) if it encounters a tag pattern for which no translation rule has been defined (and for which there is no wildcard rule or default rule), so trial and error will quickly reveal missing translation rules.

QuickStart — Five Steps to Using Xmltr

Here is a brief overview of the steps required to implement xmltr within your Frontier publishing environment. A more detailed explanation is provided in the separate xmltr tutorial.

Step 1: Download the Software

Firstly you need to download the xmltr suite:

xmltr suite as a zip archive

xmltr suite as a binhex archive

This archive contains an exported Frontier table (suites.xmltr) plus this documentation in pdf format.

You will also need the blox suite from Technology Solutions at: http://www.techsoln.com/ .

Step 2: Import the Xmltr Suite

Import the xmltr suite into your Frontier root. There is no need to run any initialisation scripts for the suite, which has no "user interface".

Some demonstrations which translate xml to HTML are provided within the "demos" table in xmltr. You can try them out and see the results they produce.

Step 3: Markup Your Source Document

Next you need to analyse the structure of your documents and design a series of tags which capture that structure. For example you might define a <page></page> tag pair to denote a web page. A <section></section> tag pair might define sections within a page. A <title> tag pair might be used for headings (and remember you can re-use the same tag names in different contexts and translate them differently). An <emph></emph> tag pair might denote text to be emphasized. And paragraphs could be delimited by <par></par>.

Edit your source document to insert the required xml tags.

See the demos in the suite or the xmltr tutorial for an example.

Step 4: Define Your Translation Rules

For each possible pattern of tags within your source document, you need to define a translation rule. For example, a <page> tag is one pattern. <page><section> might be another. <page><section><par><emph> might be yet another. Decide how you want to translate the content of each of these tags within each specific context.

You can use wildcard translation rules to define a translation rule which should be applied for a tag when the context is not important (for example, you might want <emph> translated into the HTML <b></b> tag pair irrespective of the context of the <emph> tag pair in the source document). You can also specify a default translation rule which will be used if no other rule match is found.

Translation rules can use simple string substitutions or can be arbitrarily complex Frontier scripts. Within a script translation rule, you have access (via the blox suite) to the attributes associated with the xml tag which triggered the rule. These may be used to provide additional information to the translation rule — for example a <page> tag might define attributes for the HTML file name or the template to be used when rendering the page via the Frontier WSF.

See the demos in the suite or the xmltr tutorial for examples and precise instructions for how to define translation rules.

Step 5: Translate Your Documents

Import your xml source document into Frontier, present it to the blox parser to generate an xml parse tree, then call xmltr.TranslateTree() to apply your translation rules (with whatever results they generate).

That's all there is to it.

More detailed instructions are provided in the xmltr tutorial.

Xmltr Tutorial

A step by step example using the xmltr suite to translate a simple document from xml markup into HTML for publication on a web site.

A Very Simple xml Document

For this example, we'll choose a very simple xml document which defines a single page containing a title and a paragraph with some emphasized text:

```
<page name="tutorialwebpage">
   <title>This is my home page</title>
   <section>
      <title>Introduction</title>
      <par>Page with <emph>little</emph> content.</par>
   </section>
</page>
```

(For clarity the <?xml version...> declaration at the start has been omitted.)

This particular example is included in the "demos" table in the xmltr suite.

We want to translate this xml source into an HTML page to be rendered via the Frontier Web Site Framework (WSF).

Defining Translation Rules

For this source file, we'll need to define translation rules for all the patterns of xml tags which occur. These patterns are:

```
<page>
<page><title>
<page><section>
<page><section><title>
<page><section><par>
<page><section><par><emph>
```

We define these rules by creating a **translation rule table** within the Frontier Object Database (ODB). This table structure is a hierarchy. The top level table matches each first level tag in the patterns. The second level table matches each second level tag in the patterns, and so on. Here is an outline of the table structure we need to define translations for the above tag patterns:

```
page                page content
   title            the title for a page
   section          section content
      title         the title of a section
      par           a para within a section
         emph       emphasized text in a para in a section
```

The value of each of these ODB table entries defines the **translation rule** which should be applied to translate the content which appears between corresponding tags in the xml source file. When an entry in this table refers to a subtable, the value of the entry cannot be used for the translation rule definition (since the “value” is in use for a subtable). In this case a special entry (beginning with an underscore) is used in the subtable to denote the translation rule for the parent entry. Hence, an actual Frontier table structure for the above translation rules might be:

```
page             subtable
   _page         script translation rule (for <page> tag)
   title         script translation rule
   section       subtable
      title      wpText translation rule
      par        subtable
         _par    string translation rule
         emph    string translation rule
```

The entries beginning with underscore “_” are the translation rules for the parent node — i.e. the _page entry is the translation rule for the <page></page> tag pair.

It is often the case that translation rules for such parent nodes don’t need to perform any translation (usually when all the actual content appears within child nodes). In this case, you can omit the entry beginning with the underscore, and an “identity” (or one-to-one) translation rule will be used. This can be seen in the absence of a specific rule named “_section” for the <section> tag above.

Notice that some of the translation rules are scripts while others are strings or wpTexts.

What might these translation rules look like?

String and wpText Translation Rules

String and wpText translation rules work exactly the same way, however a wpText can make editing of long translation rules easier.

In each case, the result of the translation is the text of the string or wpText in which the special token “<children/>” is replaced by the result of translating all the child nodes of the node being translated (in the xml parse table).

In the above example, the translation rules for the tag pattern:

```
<page><section><par>
```

would be evaluated by first translating all the **content** of the <par> node in the xml parse tree and then applying the translation rule for <par> to that already translated content. Translating the content of the <par> node would involve translating any tag patterns for:

```
<page><section><par><emph>
```

So in practice these two translation rules might look like this:

```
_par       <p><children/></p>
emph       <b><children/></b>
```

We're translating the xml "par" tag (but only when it occurs in the context <page><section>) into an HTML paragraph tag pair <p></p>.

The xml "emph" tag (again in this context only) is translated into the HTML **bold** tag pair <b></b>. (See the following discussion on wildcards in translation rules for performing translations when context does not matter.)

For the purposes of demonstration these translation examples are trivial. However, you can of course do more fancy kinds of translations. For example you could translate a <title> tag into a whole series of HTML instructions which not only format the title with the desired font, size and color, but also insert special spacing or a decorative graphic. Doing that kind of markup by hand in HTML would be tedious and would invite errors and inconsistency.

Script Translation Rules

Script translation rules can be used to perform much more complex translations.

When a script translation rule is invoked, the script is called and the return value (usually a string) is used as the result of the translation.

Script translation rules typically do the following:

- Translate the content of any child nodes.
- Get the values of attributes of the xml node being translated.
- Construct a result based on these.

Script translation rules can perform arbitrary processing with the full power of Frontier and the ODB at their disposal.

Each script is passed the address of the xml node being translated (i.e. the address in the xml parse tree) so that attribute values can be queried using blox.getAttribute()

Creating a new WSF page via the page translation rule script

Let's look at a script for the <page> translation rule in our above example. In this case we want to make a new web page entry in a web site table in the Frontier ODB each time we see a <page></page> tag pair. This will allow us to author multiple HTML pages in a single xml source file.

Here is a basic script to do this:

```
on _page (adrXmlNode, nodeData)
   local (adrWebpage, childString, pagename)
   # translate the content of the xml page node
   childString = xmltr.TranslateChildren(adrXmlNode, nodeData)
   # Get the “name” attribute to use as the page name in the ODB
   pagename = blox.getAttribute(adrXmlNode, “name”)
   adrWebpage = @xmltr.demos.tutorial.[pagename]
   new(wpTextType, adrWebPage)
   target.set(adrWebPage)
   wp.setText(childString)
   target.clear()
   return  “”
```

When this script is called, adrXmlNode is the address of a node in the xml parse tree corresponding to a <page></page> tag pair in the xml source document. We have used an attribute of the <page> tag (“name”) which tells us how to name the new web page object.

The call to xmltr.TranslateChildren() translates any child nodes — i.e. the full content of the page between the <page> and </page> xml tags. This is a recursive process, so translation rules will be applied to all the tag patterns in the page content and the result of all this is passed back by TranslateChildren(). The result is copied into a wpText object in the web site framework (consequently this script returns an empty string since its “result” has already been dealt with).

The web page can then be rendered in the normal Frontier manner.

In practice, such a script may do other things as well by examining and acting on other attributes of the <page> tag.

The page title needs a directive

Let’s take the above example a little further by looking at the translation rule for the <page><title> tag pattern.

Recall our xml sample file which began thus:

```
<page name=”tutorialwebpage”>
   <title>This is my home page</title>
```

The text “This is my home page” is what we want to appear at the top of the page and also in the window title bar (i.e. within the <title> tag in the HTML header). A script translation rule for this <title> tag might look like this:

```
on title(adrXmlNode, nodeData)
   local (content)
   content = xmltr.TranslateChildren(adrXmlNode, nodeData)
   return “#title \”” + content + “\”\r” + \
      “<h2>” + content + “</h2>”
```

Here we’re doing two things:

1. Making a directive which holds the text of the page title. This can be picked up in the site template to set the text used for the HTML <title> tag.
2. Generating HTML to display the title at the top of the page as an HTML “h2” heading.

Remember that this translation rule will be triggered only for a <title> tag which is contained immediately within a <page> tag. The second <title> tag which occurs in our sample xml document is within a <section> tag and so will be handled by a different translation rule.

We reuse the title tag but the result is different!

Finally we will look at a translation rule for the second <title> tag — the one which occurs in the context of a <section> tag. Referring to the table of translation rules shown earlier, we can see this was stored in a wpText. Its contents might look like this:

```
<p><font face=”arial,helvetica” color=”#0000FF” size=”+2">
<br><b><children/></b></font></p>
```

In this case we choose to translate the title of a section to a particular font size and color with a <br> for some extra space before the title. Recall that the special token <children/> is replaced with the (translated) content of the xml node — in this case the text which appears between the <title> and </title> tags.

As you can see, the <title> tag is translated differently depending on its context in the xml document. This can make markup easier. In this case we need remember only that every “component” in the xml document — page, section, subsection — can have a title and that title will be translated correctly (and differently) for each context.

Translating the xml Document

All that remains is to invoke our translation rules and view the result.

Parsing the xml

We use the blox parser to parse our xml source and generate a parse tree:

```
blox.textToXml(xmlText, @adrXmlTbl)
```

The first parameter is the xml source text (a string) and the second is the address in the ODB where we want blox to put the resulting parse tree. If no errors result from parsing the xml, we can then translate the parse tree using xmltr.

Translating the parse tree

The script which does the translation is named “TranslateTree”. You call it like this:

```
result = xmltr.TranslateTree(adrXmlTbl, adrTransTbl)
```

The first parameter is the address of an xml parse table produced from blox.textToXml() and the second parameter is the address of a translation table containing translation rules (as described earlier).

You can also provide optional parameters to TranslateTree() for more precise control over the translation process. These are detailed in the script summary.

Looking at the result

By default, the translated text is returned as the value of the TranslateTree() script.

However, this depends on the translation rules. In this tutorial example, the translation rule for the <page> tag actually intercepts the translated result for each “page” and puts it into a wpText object in the ODB. TranslateTree() in this case returns no result. For this example, the xml translated into HTML might look like this:

```
#title “This is my home page”
<h2>This is my home page</h2>
<p><font face=”arial,helvetica” color=”#0000FF” size=”+2">
<br><b>Introduction</b></font></p>
<p>Page with <b>little</b> content.</p>
```

Rendering the Web Pages

If you're using xmltr to translate xml to HTML, then you can place the result of the translation into one or more wpText objects in a Frontier website table. From that point you can render the pages using the standard Frontier Web Site Framework machinery.

Other Things You Can Do

Wildcards in translation rules

Sometimes a tag pattern occurs in many different contexts within a document, yet you want the same translation rule to apply for each context. An example might be an <emph> tag used for tagging "emphasized" text. We might want this always to translate to "bold" text wherever it occurs in the document. In this case we don't want to create repeated entries for <emph> in the translation rules table for every possible context in which <emph> might occur.

This is where **wildcard rules** come in handy. You can place one or more wildcard rules at any level of the translation rule table and an attempt will be made to match a tag pattern with these whenever an exact match does not occur. The most deeply nested wildcard rule which partially matches the given tag pattern will be used.

For example, if we place a wildcard rule for <emph> at the top level of our translation rule table, then this can match any tag pattern which **ends with** <emph>, provided there is not an exact match for that tag.

You place wildcard rules (using the same format for normal translation rules) in a subtable named "_any". This subtable effectively matches zero or more intermediate levels of tags. If this subtable is not at the top level of the translation rule table, then the "wildcard" part of the rule match will occur at that point in the tag pattern.

Default translation rules ** NEW **

You can specify a default translation rule which will be used for any tag pattern which matches neither a specific translation rule nor a wildcard rule.

This default translation rule should be named "_default". You can place default rules at any level of the translation rule table. The most deeply nested default rule which partially matches the given tag pattern will be used. To specify a default translation rule which will match **any** otherwise unmatched tag pattern, place it at the top level of the translation rule table.

Where might this be useful? Lets say you wanted to use xmltr as a "filter" to extract certain components of the xml parse tree and discard others (maybe to build a table of contents). In this case you could create translation rules to match those tag patterns you wish to extract and use a default translation rule which translates to nothing to ignore all other tag patterns.

Another use for a default rule is during development of translation rules to catch tag patterns which don't have translation rules and to translate these to a diagnostic message in your web page (rather than stopping the translation with a ScriptError).

Re-using a translation rule via "sameas"   ** NEW **

Sometimes you may want the same translation rule to be used for a number of quite different tag patterns. For example, in a page containing name and address information, you might have tags for <name>, <position>, <company>, <street> and <city>, yet you might want an identical translation (e.g. the same HTML formatting) used for each.

In this case you can provide the required translation rule for just one of the tags, say <name>, and specify that the other tags just use the same translation rule as <name>. In this example, the <position> tag would have a string or wpText translation rule which looks like this:

```
    position    sameas:<page><name>
```

This means that when a <position> tag is encountered (in the appropriate context), the translation rule for the tag pattern <page><name> will be used.

Such a translation rule should begin with the characters "sameas:" and be followed by the tag pattern (complete with xml-tag-style angle brackets) whose translation rule should be used.

When locating the translation rule for the specified tag pattern, the same pattern matching machinery is used as for normal translation rules. Hence the tag pattern appearing after "sameas:" can match a specific translation rule, a wildcard rule or a default rule.

Script rules can return data structures for special processing

The translation machinery of xmltr assumes that translation rules for each xml tag pair (or node in the xml parse tree) return strings which are then concatenated to form the complete translated result. However, within your own script translation rules, you can return something other than a string — such as a list or a table structure — provided you intercept this result and process it in the enclosing translation rule script.

Two utility scripts in the xmltr suite operate this way: CollectChildren() (which translates all the child nodes separately and puts the results in a Frontier table) and CollectSimilarChildren() (which translates all child nodes of a particular type separately and puts the results in a Frontier list). Their operation is detailed in the script summary.

Translation of entities

Xml entities are like constants in other languages — symbolic names for something. They are written with the notation: "&entityname;". The blox parser makes special parse table entries for entities and xmltr provides a special mechanism for handling the translation (or expansion) of entities.

By default, xmltr just passes entities straight through, so that they appear with the same entity notation in the translated result. If you're translating xml to HTML, this will probably be sufficient (since HTML shares the same entity notation with xml).

You can provide your own translation script for entities. It should be named "_entity" and appear in the top level of the translation rule table. Xmltr will call this script to translate each entity. Typically such a script would contain a case statement to deal with each expected entity one by one — any cases you don't want to handle you can pass to the default entity handling script provided by xmltr, TranslateEntityDefault().

The calling sequences for these scripts are provided in the script summary.

Overriding the rule finding and processing behaviours ** NEW **

You can optionally supply scripts which override the default behaviour of xmltr in two areas:

- You can provide a script which overrides the built-in rule finding script. This could be useful for example if you wanted to search more than one table of translation rules or take special actions if a translation rule is not found (instead of having a ScriptError raised).
- You can provide a script which overrides the built-in rule processing script. Use this if you want add your own special syntax for translation rules.

You can specify these override scripts when you call TranslateTree(), or you can place them in the top level of the translation rule table (with names _FindRule() and _ProcessRule() respectively).

Details are provided in the script summary.

Preserving the rule cache for faster processing ** NEW **

Xmltr builds a cache table which maps tag patterns to rules. This greatly speeds up the translation process for tag patterns which occur more than once in a document — and most do. By default this cache is recreated at the start of each translation (each call to TranslateTree()) and deleted afterwards.

You can request the cache to be preserved between calls to TranslateTree() which will speed up translations of lots of short documents. The speedup will be negligible for longer documents.

The cache is stored at the root level of the translation rule table. (Note: this is a change from xmltr version 1.0 where the cache was kept on the call stack).

Details are provided in the script summary.

Summary of Xmltr Scripts and Parameters

To Translate an xml Parse Tree

TranslateTree()

The main script you need to use is TranslateTree():

```
result = xmltr.TranslateTree(adrXmlTbl, adrTransTbl)
```

It takes two parameters: (i) an address of an xml parse tree produced by the blox parser; and (ii) an address of a translation rule table.

The return value is the result of the translation.

Actually it also takes some optional parameters as follows:

```
result = xmltr.TranslateTree(adrXmlTbl, adrTransTbl,
   userData=nil, adrFindScript=nil,
   adrRuleScript=nil, clearCache=true)
```

The optional parameters are:

- userData: Data you provide which might be required by translation scripts (for example some context information about your web site tables). You can retrieve this with GetUserData() (see below).
- adrFindScript: The address of a script which will be used instead of the built-in FindRule() script (described below).
- adrRuleScript: The address of a script which will be used instead of the built-in ProcessRule() script (described below).
- clearCache: If false, the cache of tag patterns will be preserved (in the root of the translation rule table) between calls to TranslateTree(). Speeds up translations of multiple small parse trees. Defaults to true (i.e. the cache is **not** preserved).

To Define a Script Translation Rule

Script translation rules have the following calling convention:

```
result = scriptRuleName(adrXmlNode, nodeData)
```

A translation rule script takes two parameters: (i) the address of the xml node (in the xml parse tree) currently being translated; and (ii) a reference to context data used by the translation machinery.

You can use the first parameter in conjunction with blox scripts, such as blox.GetAttribute() which will return the value of a named attribute of the xml node (i.e. an attribute associated with the corresponding xml tag pair in the source document).

The second parameter is passed through to scripts such as TranslateChildren() and is used to convey state information required in the translation process. It is also where the userData parameter to TranslateTree() is stored.

GetUserData()

Within a translation rule script you can retrieve the userData parameter supplied to TranslateTree() using the script GetUserData():

```
userData = xmltr.GetUserData(nodeData)
```

Utility Scripts for Translating Child Nodes

TranslateChildren()

```
childTable = xmltr.TranslateChildren(adrXmlNode, nodeData)
```

This script translates all the child nodes of the current xml parse tree node. adrXmlNode and nodeData are the two parameters which need to be passed to script translation rules. You will need to use TranslateChildren() within translation rule scripts to translate the “content” of a node (to which you then typically apply HTML formatting).

CollectChildren()

```
childTable = xmltr.CollectChildren(adrXmlNode,
   nodeData, nameList)
```

This script works like TranslateChildren() except that it translates only the child nodes whose names (i.e. the xml tag names) are included in nameList (which should be a list of strings). The result is returned as a table (not an address, an actual table) in which each entry is the translated content of a child node.

This script can be useful for marshalling the translated content of the nominated child nodes prior to re-assembling in a particular order for presentation, which may be different from the order in which the content appeared in the xml source file.

CollectSimilarChildren()

```
childList = xmltr.CollectSimilarChildren(adrXmlNode,
   nodeData, childName)
```

Continuing in the same vein, this script marshals the translated content of child nodes which share the same name (tag name), childName. The translated content is marshalled into a Frontier list (typically of strings, but dependent on the downstream translation rules).

Defining a Default Translation Rule

_default()

Place a translation rule with this name at any level of the translation table and it will be selected if no exact match or wildcard rule can be found for a given tag pattern. Most commonly you would place such a rule at the topmost level of the translation rule table as a “catch all” translation rule.

This could be useful, for example, if you wanted to ignore the content between any xml tags which did not have specific translation rules. In this case you would define a “_default” translation rule as an empty string.

If the “_default” translation rule is a script rule, its calling syntax as for other script translation rules is:

```
result = _default(adrXmlNode, nodeData)
```

Defining “sameas” Rules

```
rulename   sameas:<tag pattern>
```

You use a string or wpText object to define a “sameas” translation rule. In this case, the tag pattern (including angle brackets such as <page><title>) which appears after the string “sameas:” will be used to locate the actual translation rule.

When locating the translation rule for the specified tag pattern, the same pattern matching machinery is used as for normal translation rules. Hence the tag pattern appearing after “sameas:” can match a specific translation rule, a wildcard rule or a default rule.

Translation of Entities

_entity()

```
entityValue = _entity(entityName)
```

Xmltr provides a default translation for xml entities, which just preserves the xml syntax of ampersand followed by entity name followed by semicolon.

If you want to override this translation, you can provide a script at the **topmost level of the translation rule table** named “_entity”. This script will be called to translate each entity that is encountered in the xml parse tree. The parameter entityName is the string name of the entity. Normally you would include a case statement to test entityName and return a translated string.

If you want to use the default translation for any cases you don’t handle, call the built-in script TranslateEntityDefault() (described next).

TranslateEntityDefault()

```
entityValue = xmltr.TranslateEntityDefault(entityName)
```

This script provides the built-in default translation for entities (which just preserves the xml entity notation of ampersand, followed by entityname followed by semicolon).

Providing Override Scripts for Processing

_FindRule()

```
adrRule = _FindRule(nodeData, adrTransTbl)
```

You can provide your own script for finding translation rules for particular tag patterns. This script will be used instead of the built-in FindRule() script (you'll need to peruse the source of FindRule() to understand exactly how it works).

Your replacement script can be provided in two ways:

- As an optional parameter to TranslateTree().
- As a script at the topmost level of the translation rule table, named _FindRule (with the calling sequence shown above).

Typically your script might call the built-in FindRule() script and then do some additional processing. For example you could implement a hierarchy of translation rule tables by sequentially calling FindRule() on each table.

_ProcessRule()

```
result = _ProcessRule(adrXmlNode, adrRule, nodeData)
```

You can provide your own script for processing translation rules. This script will be used instead of the built-in ProcessRule() script.

Your replacement script can be provided in two ways:

- As an optional parameter to TranslateTree().
- As a script at the topmost level of the translation rule table, named _ProcessRule (with the calling sequence shown above).

Overriding the built-in ProcessRule() script could be useful if you want to implement your own special encoding of rules in the translation rule table.

Clearing the Rule Cache

ClearRuleCache()

```
xmltr.ClearRuleCache(adrTransTbl)
```

Xmltr builds a cache which maps tag patterns to translation rule addresses in the translation rule table. This more than doubles the speed of translation when tag patterns occur many times in a document (which is usually the case). By default this cache is cleared when you start a new translation with TranslateTree().

However, you can tell TranslateTree() not to clear the cache (see the description of TranslateTree() earlier in this section). In this case you may want to manually clear the cache at some stage. You can call ClearRuleCache to do this, passing the address of your translation table (the cache is stored at the top level of the translation table).

More on XSL

XSL has been mooted as “the way” to translate xml. However, XSL is a complex and as-yet-incomplete piece of work. If you’ve read the XSL spec, you may find yourself wondering whether things really need to be that complicated.

Do You Need XSL?

Although XSL is a simplification of DSSSL (the XSL spec is approx 30 pages, the DSSSL spec is several hundred pages!), it is still a complex piece of design, in part because it attempts to merge a number of pre-existing ideas (DSSSL, CSS) into a single “style language”.

An XSL document defines a set of **rules** which can be used to identify structural patterns within an XML source document. These rules then define **actions** which in turn produce a sequence of **flow objects** which define the formatting intent for the document (e.g. which font? how much indent? what line spacing? etc). But we’re not there yet. These flow objects then have to be fed into some kind of formatting engine which generates an actual formatted document.

The matching of rules to XML source constructs can be governed by **patterns** which may refer not only to particular tags within the source document but also to ancestors and descendants and attributes thereof. (The rules for rules are complicated to say the least, and one wonders how many of these “clever” ideas will ever be used in real world applications?)

XSL also has notions of **styles** to accommodate the concepts inherited from CSS (Cascading Style Sheets) and a scripting language based on JavaScript, but renamed ECMAScript.

XSL is complex because it tries to do a lot — some people might say too much.

A Simpler Alternative to XSL

For many publishing requirements, a much simpler strategy can be used which merely translates XML markup into a tool-specific markup for a particular target medium.

For the web, a translation from XML to HTML is sufficient. For print publishing a translation from XML to a markup language such as Frame Markup, PageMaker Markup or TEX macros will do the job.

Such an approach is simpler than XSL’s attempt to model document formatting in all its generality because it does a simple clearly defined job and leaves the details of formatting to existing tried and tested tools.

Further Reading

A lengthy critique of XSL by Michael Leventhal was published on the xml.com website in May 1999.

Revision History

Xmltr is an evolving tool. Here is the list of recent changes.

Initial Release, Version 1.0

Version 1.0 of xmltr was released on 22 October 1998.

What's New in Version 1.1 ** NEW **

Version 1.1 of xmltr was released on 24 January 1999.

New additions and changes in version 1.1 of xmltr are as follows:

- The documentation is now more extensive. New areas covered include the new features listed below as well as: (i) translation of entities and (ii) many more detailed descriptions of individual scripts in the script summary.
- You can now supply default translation rules which will be used if no ordinary or wildcard rule matches a tag pattern.
- You can use the "sameas:" prefix in a rule to specify that one tag pattern use the same translation rule as another tag pattern.
- You can provide scripts at the root level of the translation rule table which override built in scripts for finding or processing translation rules.
- You can preserve the rule cache between calls to TranslateTree() which can speed up translation of lots of short xml documents.

Proposed Changes to Xmltr

If you're using xmltr, please read the following announcement.

- Multiple translation tables
- Revised handling of entity translation
- Dealing with stack overflows
- Translating the root node of a parse tree
- Change to naming of special entries in translation tables
- Caching translation rules

Since xmltr was released in October 1998, I've had the opportunity to apply it in a number of website projects. From this experience has come ideas for improvements and refinements to xmltr.

In addition, a number of users of xmltr have suggested changes and improvements.

Before proceeding with a new release of xmltr, I'd like to present a summary of the changes being considered.

Some of these changes may affect existing applications of xmltr, so I'd like to hear from any users who are concerned that any of the proposed changes might cause difficulties.

Also, if there are specific changes you would like to see incorporated, please let me know.

Please address your comments to phowson@flexi.net.au

Multiple Translation Tables

Currently xmltr allows only a single translation rule table to be supplied to TranslateTree(). You can write your own custom FindRule() script to implement multiple translation tables. However, there is a case for making this part of the standard behaviour.

Proposal:

Allow a list of translation tables (addresses) to be supplied to TranslateTree() (in addition to the current option of a single translation table). In this case, xmltr would search each translation table in turn for a rule match. If only a single translation table address is supplied (as it is currently), then behaviour would be unchanged (hence backward compatibility).

Benefit:

This would allow commonly used translation rules to be factored out into a separate translation table which could be shared amongst a number translation phases or a number of websites.

Consequences:

There are potential complications arising between this proposed behaviour (multiple translation tables) and the existing options of supplying a custom _FindRule() script or custom _ProcessRule() script at the root level of a translation table (since each translation rule table could in principle have such custom scripts and their interaction could be confusing).

It is therefore proposed to remove support for custom _FindRule() and _ProcessRule() scripts at the root level of translation tables.

It would still be possible to supply custom scripts for FindRule() and ProcessRule() as optional parameters to TranslateTree() (as is currently the case).

Default rule handling would also need some adjustment under this scheme.

If you were specifying a series of translation tables, then you would place the default translation rule only in the last translation table to be searched. Otherwise a default entry in, say, the first translation table would prevent subsequent translation tables from being searched.

What I've found works well is to place the default translation rule in its own translation table which is searched last. This way you can change the default translation rule by specifying a different list of translation tables to TranslateTree(). Why might you want to do this? If you're using xmltr as a "filter" to select certain content from an xml parse tree, you'll want a default rule which ignores unwanted content (i.e. translates it to nothing). On the other hand, if you're translating the content of a web page, you might want a default rule which catches unknown tags and inserts an error message into the page.

Revised Handling of Entity Translation

Currently entity translation is handled by a custom script at the root level of a translation rule table called "_entity". This script is passed the name of an entity and should return the translation for that entity.

This scheme does not adapt to the notion of multiple translation tables, since there is no way for the script to signal that it cannot translate the given entity and that another translator "further down the line" should be given a chance to do so.

Proposal

One possible solution would be to make entity translation parallel the translation of xml tags more closely:

- Optionally allow the "_entity" entry in the root level of a translation table be a subtable (instead of a script).
- If it's a script, the behaviour is unchanged from the current behaviour. Since, under this scheme, there is no way for the script to signal whether it handled the entity, only one script would get a chance to translate the entity. This would logically be the first such script found in the list of supplied translation tables (assuming multiple translation tables as described above).

- If it’s a subtable, then the table is searched for an entry which matches the entity name. If a match is found then the value of that table entry is used as the entity translation. This could be a string (or wpText or outline) or a script (which would be called, and its return value used). If no match is found, then an entity translation table would be looked for in the next translation table in the supplied list of translation tables.
- If none of the entity translation tables handles the entity (or if there aren’t any entity translation tables) then the entity would be handed to a script provided as an optional parameter to TranslateTree().
- If no such script was provided, then a ScriptError would result, reporting the entity not translated.

Benefits:

- The ability to factor entity translation into a sequence of handlers, each associated with a translation rule table. Commonly used entity translations could be factored into a shared translation table (whose sole purpose might be to do standard entity translations).
- The ability to specify simple lookup-style entity translations using a table rather than requiring a case statement inside a script.

Consequences:

This system would remain compatible with existing behaviour when a single entity translation script is provided at the root level of a translation table.

Dealing with Stack Overflows

The most commonly reported problem with xmltr is application stack overflows, caused by deeply nested xml tag patterns and the 50 stack frame limit in Frontier version 6 and earlier.

Frontier 6.1 raises this limit to 200 stack frames, which should be ample in practice.

An alternate solution of removing recursion from parts of xmltr was considered. However, the cost in lost clarity of code would have been significant.

If stack overflows are a problem for you, consider upgrading to Frontier 6.1 or later versions.

Translating the Root Node of a Parse Tree

Currently xmltr translates only the child nodes of the supplied xml parse tree (i.e. the topmost node is ignored).

Some users have implemented a system where the web page objects in a Frontier web site refer back to a branch of an xml parse tree (via an address). When the page is rendered, that segment of the xml parse tree is translated to create the page content. This “just-in-time” translation scheme reduces the number of intermediate representations of page content which need to be stored in the Frontier ODB and streamlines the rendering process.

In this case, it is desirable to translate the topmost node of the segment of the xml parse tree.

Proposal

Add an optional parameter to TranslateTree() which causes the root node of the xml parse tree to be translated. By default the root node is not translated (current behaviour).

Benefits:

More flexibility in how translation is done.

Consequences:

This system would remain compatible with existing behaviour.

Change to Naming of Special Entries in Translation Tables

One user of xmltr (Marcel Graf) questioned the use of underscore "_" as the indicator of special translation table entries such as "_any", "_default", "_entity" and so on. He pointed out that underscore is a valid character in xml tag names and hence could create clashes if there were xml tags named "_any" or "_default" or "_entity". He suggested instead the use of forward slash "/" which is not a valid character in an xml tag name. Forward slash is the character used to identify special entries in xml parse trees within Frontier (such as those built by blox).

While I agree with the logic of this suggestion, there is a good reason why forward slash was not originally chosen.

Translation rules may be scripts and hence the name of a translation rule needs to be a valid Frontier script name (so it can be used as the name of the handler within the script outline). Forward slash is **not** a valid character in a Frontier script name unless you're prepared to write your handler as:

```
on ["/handlername"](parameters ...)
```

Proposal

While I feel that using underscore to flag special entries is the more acceptable solution the alternate scheme can be accommodated easily by making the names of special entries (currently "_any" or "_default" or "_entity") constants (strings) in the xmltr suite table.

Benefits:

Users who wish to use different names for these special entries can edit the string constants which define these special entries.

Consequences:

This system would remain compatible with existing behaviour.

Caching Translation Rules

When a match is found for a translation rule, xmltr caches the address of that rule with the corresponding xml tag pattern in a special cache table. This can speed up translation by a factor of two or more since tag patterns which occur more than once in an xml document need to be matched with a translation rule only once.

By default the rule cache table is created on the stack (i.e. as a local variable in TranslateTree()) so it is automatically discarded when TranslateTree() returns.

There has been an optional flag for TranslateTree() which causes the rule cache to be created at the root level of the translation table. This could be useful if you wish to speed up repetitive translations by reusing the rule cache.

Under a scheme which allows multiple translation tables (as described earlier), associating the rule cache with a single translation table would not make sense.

Proposal

Change the optional parameter to TranslateTree() to be a table address, which defaults to nil. If this address is nil, then the rule cache is created on the stack (the current default behaviour) and is discarded when TranslateTree() returns.

If the user supplies a table address for the optional parameter, then that table is used as the rule cache.

Benefits:

In the very rare situation where preserving the rule cache is desired, this can be accommodated. Preserving the cache or clearing it would then be the responsibility of the programmer.

Consequences:

This system would remain compatible with existing behaviour, unless you've been setting the clearCache parameter of TranslateTree() to false (in which case you'll need to modify your calls to TranslateTree()).