Markup UK

Tool-Based Transformations

Use the Markup, Luke

Liam Quin, Delightful Computing

This paper discusses the development of tools to explore and maintain XSLT, XQuery, and other markup-processing code. The tools described include FreqX, Xarcissus, and Eddie 2, which help to explore sample documents and large DTDs. The paper also covers XSLT DTD-coverage testing and encourages others to take a similar approach in developing tools to explore markup.

1. Introduction

Maybe you have to work on some existing XSLT or XQuery code, whether you wrote it years (or minutes) ago or whether someone else wrote it. Maybe you have to write new transformations or queries.

You have some exploring and some planning to do.

You need to explore the possible input. If you are lucky, maybe there is *actual* input for you to explore and, later, use for testing. Maybe there is pre-existing code to explore; after you have started work, soon there will be new code to explore!

All of these things can be done with or without tools beyond a simple text editor, an XML well-formedness checker, an XSLT or XQuery processor, and so on. But if the input is marked up, we can explore the markup, and maybe tools, whether off-the-shelf or custom, can make that much easier.

A strength of XML and especially XSLT is enabling people who do not think of themselves as programmers to do advanced text processing. It follows that you do not need to be an experienced programmer to write useful tools.

2. Project steps and phases

Although the actual break-down varies between projects and with different methodologies, the following tasks are always involved with a project involving marked-up documents:

- Requirements Analysis: What do we think we need?
- Data Analysis: What do we have?
- Strategy: How will we get there?
- Status Review: Where are we now?
- Implementation: Let’s go there!

◆ Testing and Fixes: Are we there yet?

In a waterfall model these steps were performed in order, with no feedback from one to another. This enabled fixed-priced projects but unfortunately led to systems that did not meet requirements that changed as discoveries were made or understanding was increased during development.

In an agile system, these separate tasks may be done in parallel or considered continuous. For example, data may be analyzed multiple times during a project, in different ways and with different objectives, as new needs emerge.

3. Exploring Data

The author has found that having a large number of sample input documents significantly increases the chance a project will be successful. Thousands of tens of thousands of journal articles, for example, selected across every possible journal and journal publisher involved in a project, will likely mean that almost all likely situations will actually occur, and that differences between schemas or DTDs and actual data will be discovered.

Having the *right* schemas, DTDs, entity files, and input data, and the *right* documentation is essential.

If there is sample output, it should be compared to what is generated, so having corresponding input and output is a major help.

If you are faced with exploring, say, five thousand documents, the first thing is to validate them. Clearly you don't want to lead each document by turn into an XML editor and press Validate. But a simple XSLT stylesheet might work:

```
<xsl:template match="/">
      <xsl:for-each select="uri-collection('data/*.xml')">
         <xsl:try select="doc(.)">
           <xsl:catch>{.} failed</xsl:catch>
         </xsl:try>
      </xsl:for-each>
    </xsl:template>
```

Exact details will vary depending on how the XSLT implementation handles collections; an alternative is a simple bash script that writes a list of files, and using unparsed-text-lines() to read a URI at a time, instead of using collection(), perhaps like this:

```
ls data > file-list.txt
```

or more, if escaping special characters such as & is needed.

If validating is too slow, you could use xmllint instead of Saxon on most platforms (you might need to install it). Since modern computers support multiple programs running at the same time independently, you could validate groups of sample files in parallel, again maybe with a script.

When you write small scripts in this way, make sure to put a comment at the start of each to say how to use them, and make a Makefile or a runme.sh file that runs them, to make it easy to return to this task later.

Validating files usually involves setting up an XML catalog; you might find it helpful to use **strace** as a wrapper to see which XML catalog files are being opened.

If you do this sort of basic exploration often, consider writing a simple tool. A half hour spent working on it will more than repay itself on the second project, or later in the first one. It

could be a simple shell script and take options for whether to trace files opened, and in which directory (folder) to look for data files, and which catalog to use.

This level of scripting is very easy and highly productive. Be *very* careful to remember the comments, though:

```
#! /bin/sh
  # validate all files in the data directory
  # options: -trace - turn on catalog file tracing
  TRACE=
  if test "$1" = "-trace"
  then
      TRACE="strace -e openat,open"
      shift; # delete the -trace option
  fi
  $TRACE xml-validate "$@"
```

There is no need to use 1960s big-business style comments with dates and who wrote what. If necessary, use a git repository, for example on gitlab.com, and store all your scripts there, and git will tell you who added each line and when, if you need to know.

Writing simple tools like this will increase your confidence and skills. Even if you are an experienced programmer with decades of scripting experience, the exercise will get you into the right head-space for working in the project. Think of it as like doing lunge exercises before a race!

4. Off-the-shelf tools and ad-hoc tools

After sending some time figuring out which schemas or DTDs to use (and it is usually not worth worrying too much about getting them all right at first: the client might not have supplied all of the right files or they might do database imports without validation), it's time to look inside, to look at the data.

A helpful initial question is, "what elements are used, and how often?" We could answer this for example in XPath or XQuery:

```
distinct-values(//name())
```

You might need to use something like db:open('data')//name(), depending on your XQuery or XPath processor.

The resulting list is unsorted, and doesn't tell us how often each element name occurred.

```
//name() => distinct-values() => sort() => string-join('&#x10;')
```

might give us a sorted list. We would like the numbers, too:

```
for $n in (//name() => distinct-values() => sort())
return $n || ' ' || count(//*[name() eq $n])
```

This gives ugly output like this:

```
0
a 775223
body 9669
br 9608
circle 8718
```

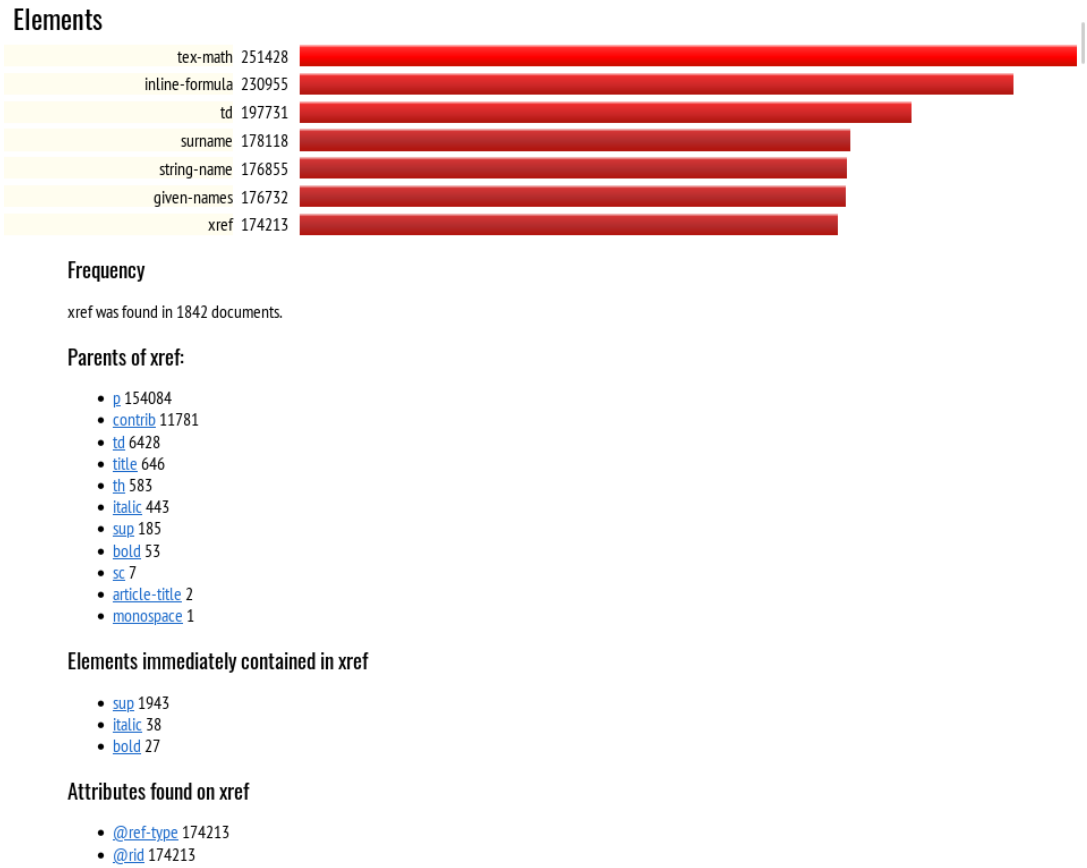
```
div 265564
font 19208
form 19232
  . . .
```

The empty name with a zero is from something without a name: the document node. We can use //*/name() instead and it will go away.

The output is still ugly. We could use some formatting, or maybe generate HTML. And after that, we might want the same for attributes. Then maybe we want to know about attribute values. With thousands of sample documents, though, there are a lot of values perhaps for ID and for dates that we might want to ignore.

You can see where this is going. Put all this into a script and comment it, give it some options or stylesheet parameters, and you can also run it on the output you generate and account for the numbers. Figure Figure 1 [5] shows what a simple HTML report might look like:

Figure 1. FreqX Report for Element Counts



At this point, an ad-hoc query has turned into a tool. Attributes were added, as per Figure 2 [6], which shows the start of the bar chart for elements sorted by frequency, once a user has clicked on, and expanded, the entry for xref. Figure Figure 3 [7] shows the attribute value table, where the user has expanded orientation to get more information.

The FreqX tool is freely available from gitlab; development was funded by Mulberry Technologies. However, it's simple enough that developing a tool to meet your own needs might be easy, as might editing the FreqX source code. If you make changes or fix bugs, please do file issues or patches, so that everyone else can benefit. For example, an often-

requested feature is being able to go from an attribute value to a list of documents in which it appears; this might be best accomplished using the eXist-db or BaseX RestXQ feature.

Figure 2. FreqX Report for Attributes

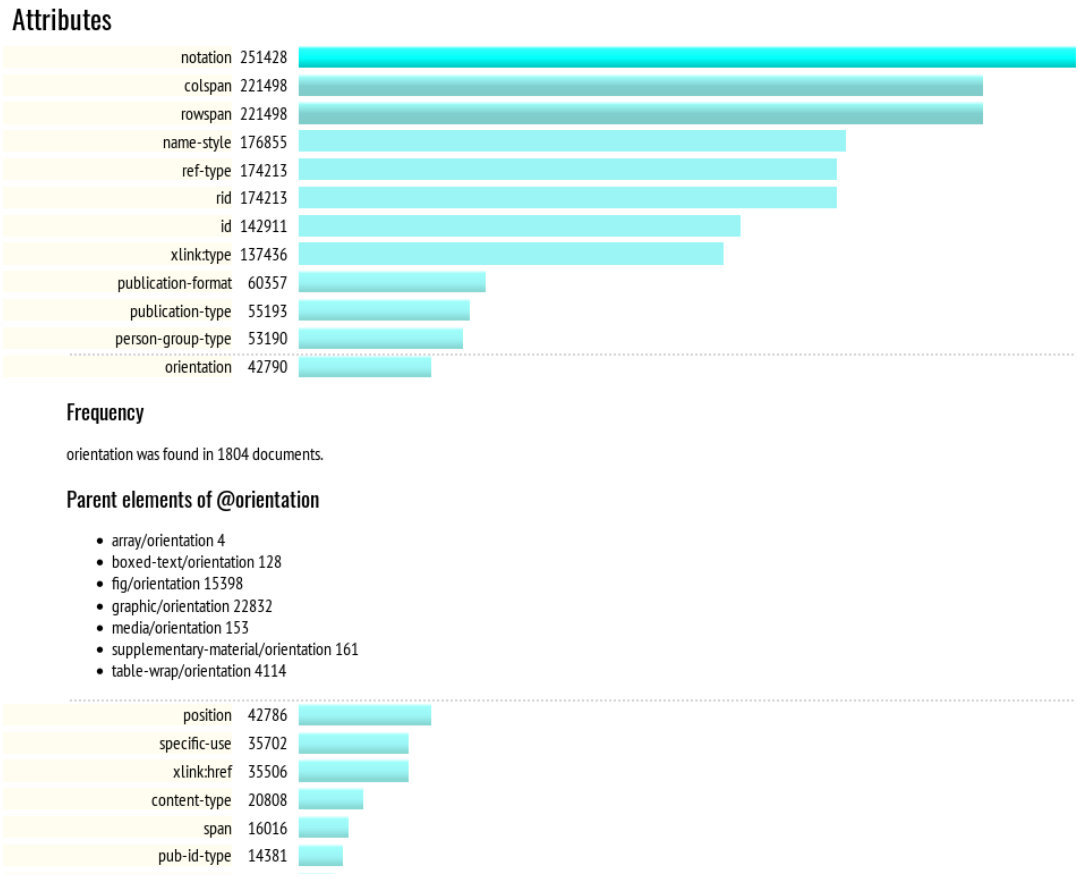
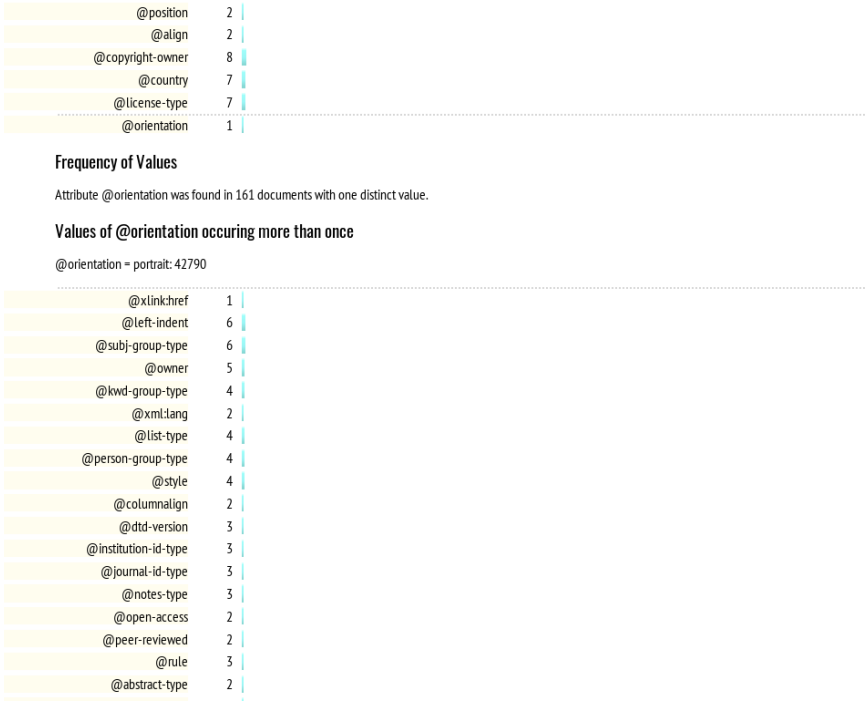


Figure 3. FreqX Report for Attribute Values



5. Analyzing Existing Code

One of the difficulties of receiving a bunch of XSLT stylesheets or XQuery files is working out what calls what. Worse, if the files contain errors or might be unfinished, or might have pieces commented out, it can be hard to process them with XML tools. Another difficulty is working out where any particular function or template is defined.

Xarcissus is a tool in two parts. the first is a scanner written in Perl, that tries to do the best it can even when the input isn't well-formed XML. The second is an XSLT stylesheet that takes the output of the scanner and makes a summary, in HTML. An earlier version also drew a dependency graph using the GraphViz library, but this turned out not to be useful in large projects, where the resulting diagram was too complex, and not to be needed in small projects.

Although Xarcissus is not currently distributed, it is freely available from the author on request. The reason for the restriction is that it's really an internal Delightful Computing tool that gets hacked around based on what's needed at any given time, rather than made into a product. For example, it can also produce Swagger (OpenAPI) files from comments in XQuery files.

Figure Figure 4 [7] shows output from Xarcissus on a moderately large XQuery project. It has found XQuery files that were referred to in HTML, in JavaScript, in XSLT, and in XQuery files. This run did not include any XSLT templates, but where those are found, their names and match patterns are recorded[11].

Figure 4. FreqX Report for Attribute Values

[1]XSLT support is incomplete at the time of writing; it seems to come and go from time to time.

The point of Xarcissus is really that you can fairly easily write a tool that is useful. You can use HTML for a simple user interface, and get something working in maybe ten minutes to an hour that will save days. Since the point of this tool is understanding, save the report it makes, maybe also save a copy of the tool with the project, and then modify the tool for the next project. Or make it into a product, with documentation!

6. Doing The Actual Work

Although Eddie 2 has been described elsewhere in some detail, the point of this paper is to describe the philosophy of making tools, so we will take a different approach, and describe its evolution.

In the first instance, the author needed to identify structural differences between two versions of the JATS DTD. This is a fairly large vocabulary, and two organizations that needed to interchange documents each had made their own variation on it.

The first version of Eddie 2 used a DTD parser module in Perl; the XML support in Python at the time did not seem to report DTD events such as finding element or entity declarations. The first version took a few hours to write and get working satisfactorily.

Once Eddie 2 was producing useful output, and could handle parameter entities more helpfully than other dtd-diff tools, the next step was to generate an XSLT stylesheet to handle each element that might appear in the input, simply copying it to its output and producing a message that it had been seen.

Including this XSLT and processing the sample documents available meant that the most frequently-occurring elements could be handled first; this meant the number of validation errors in the output reduced very quickly.

Since Eddie 2 made an HTML report, it was easy to add to this a list of elements that were in a configuration file Eddie 2 read, and to highlight whether the elements had the same content model and attributes in both DTDs, or whether they differed.

This list, shown on the right-hand side in Figure Figure 5 [8], can also be used as a sort of to-do list: any elements that differ between DTDs and are not handled in the XSLT or in the configuration file are marked with a red X. Eddie 2 also reads the XSLT file to check for coverage, and warns if there are such elements with no template to match them in the main stylesheet.

Figure 5. Eddie 2 Report

The Eddie 2 tool has proven to be highly effective in helping people write this sort of transformation between similar DTDs. Because it depends on some Perl modules, it can be a little tricky to install; a replacement written entirely in XSLT is waiting for a project to come along and fund its completion.

7. Conclusion

Writing tools is easier than it sounds, and is satisfying. You quickly build up tools that you can use, and, rather like a woodworker with a feather-board for pushing material through a table saw, the cost can be low and the benefit very high.

The tools mentioned here are (or may be) available either from gitlab.com/barefootliam or directly from the author.