Transforming Documentation from the XML Doctypes used for the Apache Website to DITA: a Case Study

Donald M Leslie
Cambridge Advanced Technology Group, IBM Research
One Charles Park
Cambridge, MA 02142
1-617-693-9981
Donald_Leslie@lotus.com

ABSTRACT
A primary factor behind the enormous interest in XML is the support it provides for transforming documents to meet the needs of information-processing applications as well as human readers working with HTML, print, and other presentation media. This case study reviews the issues we confronted, the tools we implemented, and the procedures we adopted to transform a documentation set from one XML document type to another, and from XML to HTML and Adobe PDF.

The documentation set for Xalan, the Apache XSL transformer based largely on code donated by Lotus/IBM, is written in XML, using document types shared by the projects on the Apache XML website. To present Xalan reference releases to IBM project groups, the Cambridge Advanced Technology Group has set up build procedures to transform the Xalan XML documentation to DITA, an extensible XML information typing architecture currently under development in IBM. After verifying that the DITA output conforms to its declared document type, the build publishes the DITA documentation set as HTML and as PDF.

Categories and Subject Descriptors
I.7.4 [**Document and Text Processing**]: Electronic Publishing.

General Terms
Documentation, Design, Verification.

Keywords
XML, XSL, XSLT, stylesheets, formatting objects, PDF, document transformation.

1. INTRODUCTION
In November 1999, IBM and Lotus donated two XML tools – the Xerces XML parser [6], and Xalan XSL transformer [5] – to the open-source Apache XML Project [2]. The Cambridge Advanced Technology Group, where Xalan originated under the name of LotusXSL, continues to play a major role in the development of Xalan in the Apache open-source community. The group is also responsible for periodic internal reference releases under the LotusXSL rubric to IBM product groups, such as WebSphere and Lotus Domino. Like the other members of the Apache XML project, we use a shared set of XML document types to write the Xalan documentation and a shared collection of stylesheets and tools to transform the documentation from XML to HTML.

The XML source documents are stored in the Apache repository. The HTML documents are published on the Apache XML website and distributed with the Apache releases. For the Xalan-Java and Xalan-C++ documentation, see http://xml.apache.org/xalan-j and http://xml.apache.org/xalan-c.

While work has proceeded on Xalan, an IBM cross-company workgroup including representatives from a number of IBM, Tivoli, and Lotus user assistance teams has been in the process of defining and implementing DITA [7], an extensible XML infrastructure for authoring and publishing technical documentation.

In the spring of 2001, we decided that we should use DITA to publish our internal releases. Our reasons include the following:

- Contribute to the development of DITA, a modular, flexible, and extensible XML infrastructure that is likely to be widely adopted within IBM and possibly outside of IBM as well.
- Provide user documentation in a form that other IBM groups will be prepared (as DITA is established) to work with.
- Demonstrate Xalan at work transforming from one XML document type to another, and from XML to HTML and XSL formatting objects (FO). [1]

2. GOALS
Our fundamental goal is to transform the Xalan XML source documents from the Apache document types in which they are written to a DITA XML document type. In principle, this will make it easier for IBM product groups, as they adopt DITA, to integrate all or portions of the LotusXSL/Xalan documentation with their online and printed documentation sets.

As useful as XML is for providing structural information, we do not expect our audience to read the XML in either the Apache or the DITA document types. Our Apache Xalan build procedure

SIGDOC'01, October 21-24, 2001, Santa Fe, New Mexico, USA.

[1] We use FOP [3], a sister Apache XML open-source project, to transform the formatting-objects to Adobe PDF [1].

transforms the XML sources into HTML documents. For each Apache release, we update the HTML on the Apache XML website [2].

For our internal releases, we also wanted to illustrate a server-side scenario for transforming XML documents to HTML. So in addition to publishing static HTML, we decided to set up a servlet to dynamically transform DITA XML to HTML in response to individual user requests for documents on our LotusXSL site.

In the Apache build, we also provide a procedure that assembles the XML sources into a book, transforms the result into XSL formatting objects, and uses FOP [3] to generate PDF documentation that our users can browse online and print.

Accordingly, we decided on the following primary objectives:

- Transform the Apache XML to DITA XML.
- Transform the DITA XML to HTML.
- Set up a servlet to dynamically transform DITA XML to HTML in response to user requests.
- Generate PDF documentation from the DITA XML.

In the course of developing tools to help meet these objectives, we came up with a couple of additional goals:

- Publish a utility for validating XML output.
- Publish a Xalan extension element[2] to coordinate the generation of a documentation set with a table of contents, and to streamline the piping of XML documents through a series of transformations.

3. BUILD PROCEDURES

Given that we will continue to write the documentation in Apache XML and provide both Apache Xalan builds and internal LotusXSL builds on a regular basis, we knew we needed to set up an automated build procedure for meeting our goals.

Our build tool of choice is Ant [4], an XML-based Apache open-source project. We already use Ant to build the Xalan-Java software and sample applications, to run tests, to build documentation, and to prepare our distribution files.

To create an Ant build, you set up an XML <project> file with a <target> element for each task. To meet our goals, we defined two basic tasks.

The first task transforms Apache to DITA, validates the output, assembles an HTML table of contents for the Xalan documentation set, and optionally transforms DITA to HTML.

The second task merges the DITA documents generated by the first task into a book structure, transforms the result to XSL formatting objects, and uses FOP to generate a PDF file.

[2] An extension element is a custom "instruction" element placed in a stylesheet template. XSLT provides a syntax for instantiating extension elements, but the implementation of a given extension element is vendor specific, so only Xalan can execute Xalan extension elements, which may be implemented in Java or one of several scripting languages. See "Extensions" in Xalan [9].

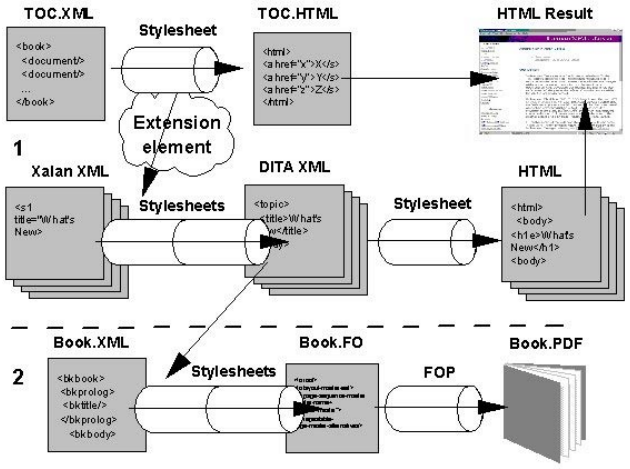

Figure 1. Build tasks.

4. APACHE TO DITA

The first build task applies an XSL stylesheet to an XML document that conforms to the Apache <book> document type. It contains a title, copyright information, and a <document> element for each of the XML source documents in the Xalan documentation set.

In our standard Apache build, this file is used with a tool called Stylebook and a collection of stylesheets to transform the XML source documents to HTML and to place a table-of-contents in each of the HTML output files.

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "../style/dtd/book.dtd">

<book title="User's Guide"
      copyright="2001 The Apache Software Foundation">

  <document id="ltsxslintro"
        label="LotusXSL 2"
        source="lotusxsl/ltsxslintro.xml"/>

  <document id="whatsnew"
        label="What's New"
        source="xalan/whatsnew.xml"/>

  <document id="dtm"
        label="DTM"
        source="xalan/dtm.xml"/>
  ...
</book>
```

Figure 2. Apache <book> document

The new Apache-to-DITA build procedure applies a stylesheet to this <book> document. This stylesheet generates a table of contents for the Xalan documentation set. For each source document, it uses an XSL extension element in conjunction with two other stylesheets to perform a transformation to DITA XML. The extension element and these stylesheets perform the real work.

The build then uses the Validate utility to verify that each DITA document conforms to the DITA <topic> document type. We implemented the extension element and the Validate utility in Java.

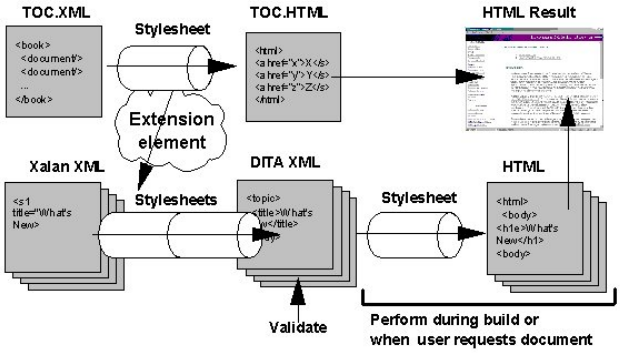

Figure 3. Apache to DITA to HTML

4.1 Setting up a Transformation Pipeline

To transform each XML source file from Apache to DITA, we perform two transformations. The first transformation modifies the basic structure from a hierarchy of sections to a collection of topics, some topics nested in other topics (see section 4.2.1 for the details). The second transformation does all the work of transforming individual Apache elements into the appropriate DITA elements.

Initially, we performed two separate transformations. That is, we wrote the output of the first transformation to a file, then used that file as input for the second transformation. But we had no need for the intermediate file, which is neither Apache nor DITA. So we decided to write a Xalan extension element to directly pipe the output of the first transformation into the second transformation without writing this throwaway file.

We began by writing a Xalan extension element designed to handle precisely the transformations we knew are required to transform each Apache XML document to DITA. The extension element obtained information for each transformation from stylesheet parameters and from various attributes in the current context node (the <document> element that provides the context for each transformation) in the base stylesheet.

It was clear that if we were to alter the structure of the XML input <book> document or the stylesheets and the input parameters they require, we would have to reimplement the extension element as well. In other words, our extension implement was not generic or portable.

But we realized that we were dealing with a very common setup:

- A list of documents to be transformed.
- A stylesheet for turning that list into an HTML table of contents.
- One or more additional stylesheets for transforming each source document to the desired HTML result. As mentioned above, we use two stylesheets and two transformations to transform from Apache XML to DITA.

Accordingly, we came up with a pipeDocument extension element supported by a much more generic implementation

The pipeDocument extension element contains attributes identifying the source and the output, and a <stylesheet> element for each transformation we need to perform. In turn, each <stylesheet> element contains <param> elements for any stylesheet parameters that need to be set.

```
<xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            xmlns:pipe="xalan://org.apache.xalan.lib.PipeDocument"
            extension-element-prefixes="pipe">          \ Extension element
<xsl:param name="xml-dest"/>                                namespace and prefix
...
<xsl:template match="document">                         — Generate TOC entry
  <a href="{@id}.html"><xsl:value-of select="@label"/></a>

  <pipe:pipeDocument source="{@source}"                 — Extension element
              target="{$xml-dest}/{@id}.xml">
    <stylesheet href="xalanprep.xsl">
      <param name="doc-id" value="{@id}"/>
    </stylesheet>
    <stylesheet href="xalanprep2dita.xsl"/>
  </pipe:pipeDocument>

</xsl:template
...
</xsl:stylesheet>
```

Figure 4. pipeDocument extension element

Our extension element gets the source document from the <document> element source attribute, the path to the output directory from the stylesheet xml-dest parameter, and uses the <document> element id attribute to set a stylesheet parameter for the first transformation.

For detailed information about stylesheet syntax, including extension elements, see the XSL Transformations specification [12] and Kay [8] or Tidwell [10]. For more information about the Xalan pipeDocument extension, see "Extensions Library" in Xalan [5].

4.2 Transformation Details

The document type used for most Xalan documents is organized in terms of sections and subsections. Each document is a top-level section (an <s1> element) that contains a variety of content elements (paragraphs, lists, tables, source code, and so on) and subsections (<s2> elements). Each <s2> element in turn may contain the same content elements and <s3> subsections. The document declaration carries this organization down to <s4>. Each <s*n*> element contains a title attribute defining the section title.

The core DITA document type, on the other hand, is <topic>. A <topic> contains an optional <prolog> element, a <title> element, and a <body> element with elements containing the topic content. A <topic> may also contain one or more nested topics. When this occurs, the nested topics are placed after the <body> element, not within the <body> element of the containing <topic>.

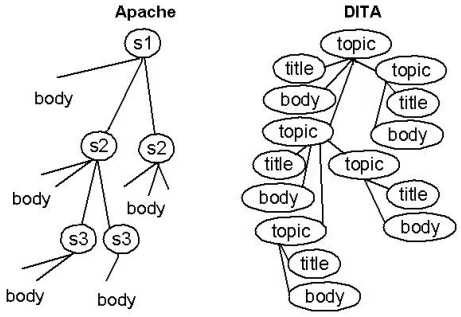

Figure 5. Apache and DITA document trees

Whereas an Apache <s1> document is a structured document with up to four section levels, a DITA <topic> is a self-contained building block that may stand alone or be combined hierarchically and serially with other building blocks to form a document.

4.2.1 Separating a nested topic from the body of its containing topic

The first challenge in transforming from <s1> to <topic> is to separate each nested topic from the body of its containing topic. To accomplish this end, we set up a stylesheet template to apply to all <s1>, <s2>, <s3>, and <s4> elements. In this template, we create a <topic> element with a <title> and <body>. We use the xsl:copy-of instruction with an XPath expression to copy everything into the body except any nested <s2>, <s3>, or <s4> elements. The XPath expression selects all nodes that are not named s2, s3, or s4. The stylesheet recursively applies the same template to the remaining <s2>, <s3>, and <s4> elements.

```
<xsl:template match="s1|s2|s3|s4">
  <topic id="...">
    <title><xsl:value-of select="./@title"/></title>
    <body>
      <xsl:copy-of select="node()[not(name()='s2')
                          and not(name()='s3')
                          and not(name()='s4')]"/>
    </body>
    <xsl:apply-templates select="s2|s3|s4"/>
  </topic>
</xsl:template>
```

Figure 6. Copy Xalan sections to DITA topics

Our Apache documents conform to the editorial convention (not imposed by the <s1> doctype) that the body of each section must precede any subsections, so the transformation performed in Figure 6 does not alter the original information flow.

4.2.2 Establishing <topic> ids

Each DITA <topic> requires an id attribute, a unique identifier that can be used to define links. For simplicity, Figure 6 ignores the procedure for setting <topic> ids. The <s1> – <s4> elements have no such attribute. Each <s1> document is its own file, so the filename can provide an id for its <s1> element. Most of the <s2>, <s3>, and <s4> elements are preceded by an <anchor> element with a name attribute, so we can simply get the value of this attribute and assign it to the id of the following <topic>. For any <s2> – <s4> elements not associated with an <anchor> element, we can use the XSLT generate-id() function to generate a unique id.

In the stylesheet, we pass in a doc-id parameter with the document file name and use an xsl:choose structure with xsl:when statements to assign the ids.

```
<xsl:param name="doc_id"/>
...
<xsl:when test="name()='s1'">
 <topic id="{$doc-id}">
...
<xsl:when test="preceding-sibling::*[1][name()='anchor']">
  <topic id="{$doc-id}_{preceding-sibling::anchor[1]/attribute::name}">
...
<xsl:otherwise>
  <topic id="{$doc-id}_{generate-id()}">
...
```

Figure 7. Setting <topic> ids

4.2.3 Piping the output of the first stylesheet to a second stylesheet.

The first stylesheet called by the extension element generates correctly nested topics. Each <topic> has an id attribute, a <title> element, a <body> element, and may contain nested topics, each of which displays the same structure. The content of each <body> element, however, still has the same structure as in the original Apache XML.

The pipeDocument extension element pipes the output from the first transformation to a second stylesheet, which transforms individual Apache elements, such as tables and images, into their DITA equivalents. A number of elements, such as paragraphs, and single-level lists, are simply copied from the intermediate result generated by the first stylesheet to the DITA output. As the following section explains, nested lists require special treatment.

4.2.4 Handling nested lists

In the Apache s1 doctype, a list element may contain a nested list. The nested list is a child of the <ol> or <ul> parent list, and a sibling of the parent's list items (<li> elements). The DITA topic doctype, on the other hand, requires a nested list to be a child of one of the list items in the parent list. As you can see in Figure 8, the difference comes down to where a list item is closed.

Apache

```
<ol>
  <li>item 1</li>
  <ul>
    <li>nested 1</li>
    <li>nested 2</li>
  </ul>
  <li>item 2</li>
  <li>item 3</li>
</ol>
```

DITA

```
<ol>
  <li>item 1
    <ul>
      <li>nested 1</li>
      <li>nested 2</li>
    </ul>
  </li>
  <li>item 2</li>
  <li>item 3</li>
</ol>
```

Figure 8. A Nested list

For any <li> element for which the following sibling is an <ol> or <ul> element, copy the following sibling and its children before closing the <li> element.

Figure 9 contains the stylesheet templates responsible for accomplishing the structural adjustment illustrated in Figure 8.

```
 <!–List element followed by a nested list -->
<xsl:template
  match="li[following-sibling::*[1][name()='ul'
                    or name()='ol']]" mode="copy">
  <li><!-- start the list element –>
    <xsl:apply-templates mode="copy"/>
    <xsl:call-template name="nested-list">
      <!-- param contains the nested list -->
      <xsl:with-param name="nest" select="following-sibling::*[1]"/>
    </xsl:call-template>
  </li><!-- close the list element -->
</xsl:template>

<!-- Put the nested list in the list element -->
<xsl:template name="nested-list">
  <xsl:param name="nest"/>
    <xsl:element name="{name($nest)}">
      <xsl:apply-templates select="$nest/*" mode="copy"/>
    </xsl:element>
</xsl:template>
```

```
<!– if a list is nested, it has already been copied into
    a list element . Don't copy it again.–>
<xsl:template match="ul|ol" mode="copy">
  <xsl:if test="not(preceding-sibling::*[1][name()='li'])">
    <xsl:element name="{name(.)}">
      <xsl:apply-templates mode="copy"/>
    </xsl:element>
  </xsl:if>
</xsl:template>
```

Figure 9. Rearranging nested lists

5. VALIDATION

After the transformation process from Apache XML to DITA XML has been completed for all the source documents, the Ant build uses a utility to validate the DITA results.

The Validate utility does the following:

- Uses the JAXP [9] interface to get a SAXParserFactory.
- Sets validation and namespace support to true.
- Instantiates a SAXParser.
- Sets up a SAX event handler to verify that each document contains a DOCTYPE declaration and to listen for errors or warnings.
- Parses the XML output

If there is a DOCTYPE declaration and no errors or warnings occur during the parse, the document is valid. This procedure works for any set of XML files containing DOCTYPE declarations.

We wrote a general-purpose Validate utility that can be instructed to check an individual XML file or all the XML files in a directory, and to report the results to the screen or to a log file. For the details, see "Samples" in Xalan [9].

In our preliminary stylesheet implementations, Validate revealed that we were not handling nested lists as the DITA topic document type required. When we had revised our transformation of nested lists (see section 4.2.4), Validate verified that we were now meeting the DITA topic requirement for nested lists.

```
dita.validate:
    [echo] dita.validate calling Validate class
           on ./build/dita/xml4html
    [java] VALID commandline.xml
    [java] VALID design2_0_0.xml
    [java] VALID dtm.xml
    [java] VALID extensions.xml
    [java] VALID extensionslib.xml
    [java] VALID faq.xml
    [java] VALID getstarted.xml
    [java] VALID ltsxslintro.xml
    [java] VALID overview.xml
    [java] VALID readme.xml
    [java] VALID samples.xml
    [java] VALID trax.xml
    [java] VALID usagepatterns.xml
    [java] VALID whatsnew.xml
    [java] ================SUMMARY=========================
    [java] Parsed 14 .xml files in ./build/dita/xml4html.
    [java] 14 files are valid.
```

Figure 10. Ant build Validate output

6. SERVE HTML OR XML?

As we do for Apache, we want to make the HTML documentation available to our users on our website (a company-wide intranet site in this case), as well as include the HTML with our distribution files. But we also want use a servlet to transform DITA XML to HTML in response to user requests for documents.

Why bother? A couple of motivations come to mind.

1. Reliance on server-side transformations means that you can be sure users are getting the latest documentation based on the current state of your XML sources. You are not required to refresh the website each time a change occurs.
2. Given that the servlet has access to the XML with the structured information it contains, in theory, it could do more than perform a simple transformation to HTML. It could, for example, respond to a more complex request by assembling the XML from multiple source documents and/or database tables, and then transform the result.

In light of these potential objectives, we wanted to demonstrate to our users a simple strategy for performing transformations on the server.

6.1 Importing the DITA Stylesheet

The DITA workgroup has established a basic stylesheet for transforming generic DITA to HTML. From the doctype perspective, the DITA we generate is generic, so in principle, we can use this stylesheet. In practice, however, there are some minor differences, mostly having to do with links. For example, if we are serving the DITA XML with a servlet that transforms it upon request, we need to convert the internal links to servlet calls. If we are serving static HTML, we need to transform the ".xml" in our internal targets to ".html".

Figure 11. Links

Accordingly, we use a stylesheet that imports the standard DITA stylesheet. The imported stylesheet does most of the work. All we do in our stylesheet is provide special behavior where required. In other words our DITA is generic (no specialization), but we "specialize" the DITA stylesheets when we publish the Xalan documentation for our readers.

6.2 Transforming XML to HTML with a Servlet

We set up a servlet to respond to requests by transforming the XML to HTML and returning the HTML.

Xalan-Java ships with a number of sample servlets, one of which is particularly apt. The servlet request includes the XML document to be transformed and optionally an XSL stylesheet to perform the transformation. If the XSL stylesheet is not included (in the servlet request or as a servlet processing instruction in the XML document to be transformed) the servlet uses a pre-compiled stylesheet that it has been instructed to compile and load when the servlet is launched.

When multiple documents are to be transformed with the same stylesheet (a common case), this arrangement improves performance by avoiding the overhead of reprocessing the stylesheet for each transformation. All that is required is to set up the table of contents and each of the internal links to call the servlet

with a parameter identifying the document to be transformed (see Figure 11).

6.3 HTML Output

As you can see in Figure 12, the HTML output we generate from DITA is somewhat different in appearance than the HTML output generated by our Apache build.

For the Apache website, we generate HTML that conforms to the Apache XML Project presentation standards. For our IBM customers, we use a different stylesheet to conform to their expectations. Stylesheets give us the freedom to generate different presentations from the same source and similar presentations from different sources.

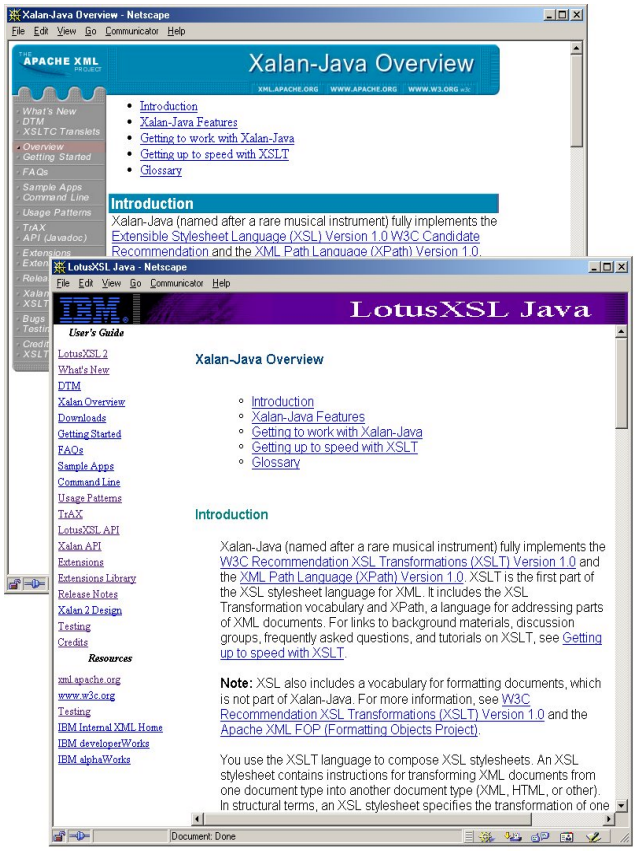

Figure 12. Apache and DITA HTML

7. GENERATING PDF

The basic methodology for turning XML documents into PDF is to use a stylesheet processor to transform the XML into an XML tree of formatting objects, and then to use an XSL print formatter to process the formatting objects tree and generate a PDF file.

Given that we wanted to print a collection of XML documents as a single book, we set up a an Ant build task to do the following:

1. Assemble the XML documents into an XML book.
2. Transform the book to an XSL formatting-objects tree (an FO file).
3. Turn the FO file into a PDF file.

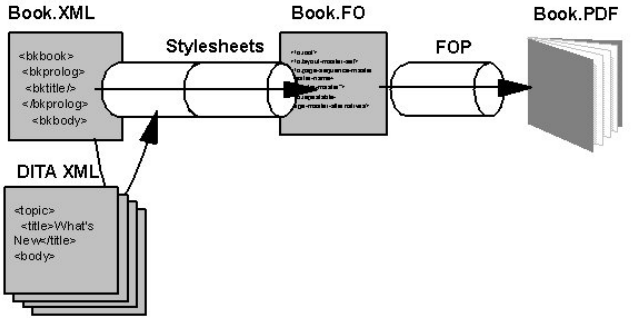

Figure 13. DITA to PDF

7.1 Assembling the book

As already noted, we use a book document to specify which XML documents are to be transformed to HTML and to put a book title, a table of contents, and a copyright notice in each HTML output document. This document uses the <book> document type defined for the Apache XML projects.

The DITA workgroup has defined a <bkbook> document type for purposes of generating printed output. It is more extensive than the Apache <book>. The fundamental difference is that <bkbook> includes structural elements mapping individual topics to structural book components, such as front matter, book parts, chapters, sections, several levels of subsections, appendixes, and so on.

Figure 14 contains part of the LotusXSL book document. After the front-matter material, the body (<bkobdy>) contains a number of <bkchapter> elements, each with a reference to an XML document.

```
<?xml version="1.0"?>
<!DOCTYPE bkbook SYSTEM "dtd/bkbook.dtd">
<bkbook>
  <bkprolog>
    <bktitle>LotusXSL-Java version 2.2</bktitle>
    <bksubtitle>XSL Transformer</bksubtitle>
    <bkauthor>Don Leslie</bkauthor>
    <bkpubdate>July 2001</bkpubdate>
    <bkcopyr>Apache Software Foundation, 2001</bkcopyr>
  </bkprolog>
  <bkbody>
    <bkchapter><topicref href="ltsxslintro.xml"/></bkchapter>
    <bkchapter><topicref href="whatsnew.xml"/></bkchapter>
    <bkchapter><topicref href="dtm.xml"/></bkchapter>
    <bkchapter><topicref href="overview.xml"/></bkchapter>
    <bkchapter><topicref href="getstarted.xml"/></bkchapter>
    ...
  </bkbody>
</bkbook>
```

Figure 14. LotusXSL <bkbook> document

Each XML document contains up to four levels of nested topics. The top-level topic is a <bkchapter> topic. Second–level topics need to be mapped to <bksection> elements, third-level topics to <bksubsection> elements, and fourth-level topics to <bksubsect1> elements.

We accomplish this mapping by applying a stylesheet to the preceding <bkbook> document, which pulls in each <bkchapter> document, inserting <bksection>, <bksubsection>, and <bksubsect1> elements, and placing topics within those elements: the topics are no longer nested within each other. The stylesheet template uses nested xsl:for-each instructions to navigate through each document, and xsl:copy-of instructions to pull in the topics.

The input and output documents of this transformation are both <bkbook> documents. Whereas the input (Figure 14) is a shell, the output (Figure 15) contains the entire book.

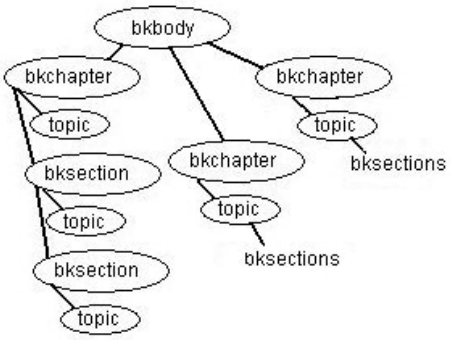

Figure 15. <bkbook> output tree

7.2 Transforming the book to formatting objects

The Extensible Stylesheet Language specification [11] for formatting objects is still a work in progress, as is the FOP implementation, so our work in this area is still tentative in nature.

The DITA workgroup has set up a stylesheet to perform this transformation. Our task is conceptually similar to the task of transforming the DITA XML output to HTML: extend the DITA standard stylesheets where necessary to override or add a particular transformation.

As with the DITA transformation to HTML, the most important overrides were simply to use existing topic and paragraph ids instead of generating new ids, so that internal links work.

7.3 Using PipeDocument to package these transformations

In our first implementation, we made two calls to the Xalan command-line utility: the first to merge the DITA sources into the expanded <bkbook> document, and the second to transform the <bkbook> output tree into a tree of XSL formatting objects.

Once we had created our general-purpose pipeDocument extension element, we could use it to perform this operation in a single step. We simply place the extension element in an otherwise empty stylesheet. Figure 16 displays the stylesheet we use to transform the bkbook shell to the formatting objects tree.

The Ant build calls the Xalan transformer with this stylesheet and the input parameters for source and target. The extension applies the first stylesheet specified in the pipeDocument element (printbook_assemble.xsl) to the <bkbook> document in Figure 14, and the second stylesheet (bkbook8x11_xalan.xsl) to the output of the first transformation (the merged book in Figure 15), outputting the result to a formatting objects file.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:pipe="xalan://org.apache.xalan.lib.PipeDocument"
                extension-element-prefixes="pipe">

<xsl:param name="source"/>
<xsl:param name="target"/>

<xsl:template match="/">

 <pipe:pipeDocument      — Extension element
     source="{$source}"
     target="{$target}">
   <stylesheet href="printbook_assemble.xsl"/>
   <stylesheet href="bkbook8x11_xalan.xsl"/>
 </pipe:pipeDocument>

</xsl:template>

</xsl:stylesheet>
```

Figure 16. Pipe DITA XML to FO

7.4 Generating PDF

We use Apache FOP [3] to transform the tree of formatting objects into a PDF file.

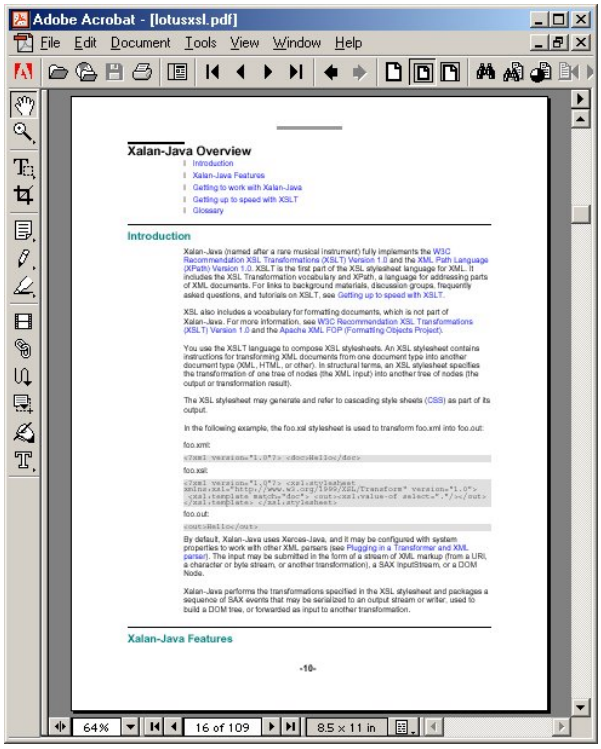

Figure 17. PDF from DITA

Using FOP is quite straightforward. You simply download and expand the binary release file, add a couple of JARS in the download to the classpath, and call the FOP command-line utility.

Like the specification it implements, FOP is not complete, and the transformation does not yet produce print of the quality you would expect to find in a commercially printed book. But the output is certainly usable. In addition to printing it, users can read it online, taking advantage of its internal and external links. The external links are to Javadoc and other HTML files on the Apache and other internet sites.

8. CONCLUSION

This case study demonstrates that it is feasible to set up an automated process to transform a collection of documents from one XML document type to another. This same process may be extended to also generate HTML and PDF output for readers. A servlet may be used to transform XML to HTML on the server in response to user requests.

In the course of carrying out this case study, we recognized a common pattern involved in transforming a set of XML documents to a set of HTML documents with a shared table of contents. To simplify this process we implemented support for a pipeDocumdent extension element that can be embedded in the stylesheet that generates the table of contents and used to perform the transformations required to transform each source document to the desired output.

We used this same pipeDocument extension element in an otherwise empty stylesheet to merge our DITA XML documents into a book and transform the book into XSL formatting objects to provide input for generating a PDF document.

We also created a simple Validate tool to verify that our XML output conforms to its specified document type.

As XML usage accelerates, we believe other documentation groups and website managers can exploit and extend the preliminary infrastructure we have created in our endeavor to transform the Xalan documentation set from Apache XML to DITA XML and from DITA XML to HTML and PDF.

9. ACKNOWLEDGMENTS

10. REFERENCES

[1] Adobe PDF (Portable Document Format). http://www.adobe.com/products/acrobat/adobepdf.html

[2] Apache Software Foundation, Apache XML Project. http://xml.apache.org/index.html.

[3] Apache Software Foundation, FOP. http://xml.apache.org/fop/index.html

[4] Apache Software Foundation, Jakarta Ant. http://jakarta.apache.org/ant/index.html

[5] Apache Software Foundation, Xalan Java. http://xml.apache.org/xalan-j/index.html

[6] Apache Software Foundation, Xerces Java Parser. http://xml.apache.org/xerces-j/index.html

[7] Day, D. R., Priestley, M., and Schell, D. A. Introduction to the Darwin Information Typing Architecture. IBM developerWorks, 2001, http://www-106.ibm.com/developerworks/xml/library/x-dita1/index.html

[8] Kay, M. XSLT Programmer's Reference. Wrox Press Ltd., Birmingham, UK, 2000.

[9] Sun Microsystems, Inc., Java API for XML Processing (JAXP) 1.1 Public Review 2. http://java.sun.com/aboutJava/communityprocess/review/jsr063/jaxp-pd2.pdf.

[10] Tidwell, Doug, XSLT. O'Reilly & Associates, Sebastopol, CA, 2001.

[11] W3C, Extensible Stylesheet Language (XSL) Version 1.0. http://www.w3.org/TR/xsl/.

[12] W3C, XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt/.