Docbook In ConTEXt, a ConTEXt XML Mapping for DocBook Documents

Simon Pepping
Elsevier, Amsterdam, Netherlands
s.pepping@elsevier.com, spepping@leverkruid.nl
http://www.leverkruid.nl/

Abstract

Docbook In ConTEXt is a ConTEXt module that allows one to produce a typeset version of a Docbook XML file, in dvi or pdf format. It takes a Docbook XML file as input for TEX. ConTEXt's built-in XML parser parses the file and maps opening and closing tags to the ConTEXt commands specified in the *Docbook In ConTEXt* module.

The first part of this article describes how one can run ConTEXt on a Docbook XML file using the *Docbook In ConTEXt* module. The second part deals with some aspects of programming this module. It presents the general framework, discusses some of the problems encountered, and highlights the programming of some noteworthy elements.

Résumé

Docbook In ConTEXt est un module ConTEXt qui permet la production d'une version composée d'un fichier XML Docbook, aux formats DVI ou PDF. Il prend un fichier XML Docbook en entrée de TEX. Le parseur XML interne de ConTEXt analyse le fichier et associe les balises ouvrantes et fermantes aux commandes ConTEXt spécifiées dans le module *Docbook In ConTEXt*.

La première partie de cet article décrit comment on peut lancer ConTEXt sur un fichier XML Docbook en utilisant le module *Docbook In ConTEXt*. La deuxième partie traite de certains aspects de programmation de ce module. Nous y présentons le cadre de travail général, nous discutons certains problèmes que nous avons rencontrés, et nous soulignons la programmation de certains éléments dignes de considération.

What is Docbook In ConTEXt?

Docbook In ConTEXt combines two technologies that are widely used by authors of technical literature: the Docbook DTD and the ConTEXt macro package for TEX.

It is a ConTEXt module that allows one to produce a typeset version of a Docbook XML file, in dvi or pdf format.

It takes a Docbook XML file as input for TEX. ConTEXt's built-in XML parser parses the file and applies ConTEXt commands when it reads opening and closing tags. Which ConTEXt commands are applied, and therefore how the output is formatted, is determined by the Docbook In ConTEXt module.

XML, Docbook and stylesheets Docbook documents are XML articles. They contain XML tags, such as `<title>` and the corresponding end tag `</title>`. These two tags mark the enclosed text as the title of the document. This is rather similar to ConTEXt's `\title` command. The difference is that XML more precisely prescribes which information is tagged, and which names are used for the tagging. This is defined in the DTD. For each document an XML author is free to select a suitable DTD, write a new DTD, or go the way of free structuring and do without a DTD.

Docbook is a large DTD for technical literature, books and articles. It defines the possible structure for such documents. When an author chooses to structure his document according to the Docbook DTD, it can be processed with Docbook utilities. Examples are such utilities as stylesheets like Docbook In ConTEXt, or applications which extract information from the document, like the title and the author names.

The example document in fig. 1 shows the structure of a small Docbook article. Everything is between the `article` start and end tags, which indicates that it is an article and not a book or even a set of books. The first part is the `articleinfo`, with the metadata: title and authors. One could add much more information, such as affiliations, revision history, abstract, copyright, etc. Next comes the main text. It consists of sections with titles, subsections and paragraphs. The text in the paragraphs is less structured. It consists of unmarked text, in which special parts are marked, e.g. literal texts, file names, program listings. In XML jargon this is called 'mixed content'. In Docbook one marks especially those parts that are relevant in technical literature. In a history book one would want to mark a different set of notions, and one would therefore use a different DTD defining

```
<?xml version="1.0" ?>
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
                         "docbookx.dtd" []>
<article>

<articleinfo>
<title>DocBook In ConTeXt, ConTeXt XML mapping for DocBook
documents</title>
<authorgroup>
<author>
<firstname>Simon</firstname>
<surname>Pepping</surname>
</author>
<author>
<firstname>Michael</firstname>
<surname>Wiedmann</surname>
</author>
</authorgroup>
</articleinfo>

<section>
<title>Installation</title>
<para role="first">Change directory to the top directory of one of the
<literal>texmf</literal> trees of your TeX installation,
e.g. <filename>/usr/share/texmf</filename>, and
<command>untar</command> the distribution file
<filename>DocbookInContext.tar.gz</filename>. Then run the command
<command>mktexlsr</command> for that tree, e.g. <command>mktexlsr
/usr/share/texmf</command>.</para>
</section>

<section>
<title>Usage</title>
<programlisting>
\input xtag-docbook
\setupheadertexts[section][pagenumber]
\setupheader[leftwidth=.7\hsize,style=slanted]

% customizations
\setuphead[section][style=bia,number=no,align=right]
\setupepigraph[narrower={1*right},command=\bi]
\setupattribution[command=---]
\setupXMLDBlists[notoc]
\setupXMLDB[background=off]
\def\XMLDBarticleinfotitle#1%
    {\startalignment[middle]\bib #1\stopalignment\blank[1*big]}
\defineXMLattributeaction[para][role][first]{\bf}
</programlisting>
</para>
</section>

</article>
```

Fig. 1: Example of a Docbook article.

DocBook In ConTeXt

Simon Pepping
Michael Wiedmann

Contents

1 Installation

There was things which he stretched, but mainly he told the truth.
— Mark Twain, Huckleberry Finn (1884)

Change directory to the top directory of one of the `texmf` trees of your TeX installation, e.g. `/usr/share/texmf`, and `untar` the distribution file `DocbookInContext.tar.gz`. Then run the command `mktexlsr` for that tree, e.g. `mktexlsr /usr/share/texmf`. Test the result by issuing the command `kpsewhich xtag-docbook.tex`. The reply should be the path of one of the files just installed, e.g. /usr/share/texmf/tex/context/DocbookInContext/xtag-docbook.tex.

2 Usage

Run context on a Docbook XML file using `xtag-docbook.tex` as the map file. Context offers several possibilities to do this, see the context XML manual `example.pdf`.

One option is to construct a TeX file that inputs the mapping file `xtag-docbook.tex`, and in the text block inputs the XML file with the command `\processXMLfilegrouped`. For example:

```
\input xtag-docbook
\setupheadertexts[section][pagenumber]
\setupheader[leftwidth=.7\hsize,style=slanted]
\setuppagenumbering[location=]
\setupitemize[each][packed][before=,after=,indentnext=no]

\starttext
\processXMLfilegrouped{\jobname.xml}
\stoptext
```

Fig. 2: The article of fig. 1 as formatted by ConTEXt with the Docbook In ConTEXt macros.

suitable tags. In the text one may also insert index terms. At the end of the document one could add appendices, bibliographies, etc.

It is the task of a stylesheet like Docbook In ConTEXt to pick up the tags and render their content with appropriate formatting. Program listings are rendered in a monospaced font, and the line layout is preserved. Index terms are saved and used to construct the index. When one looks at the `articleinfo`, one sees that the XML document does not always contain punctuation and spacing. There is no punctuation between the authors. There may be white space between the first name and surname, but it may also be absent. The stylesheet should add punctuation and white space to the rendering as required. Fig. 2 shows how the small document in fig. 1 is rendered by Docbook In ConTEXt.

The Docbook[1] DTD has been available since the early 1990s. Over the years it has evolved into an extensive DTD for all technical literature. Later in the 1990s extensive, customizable stylesheets[2] became available, written in DSSSL. The Jade program, the JadeTEX macro package (and the numerous underlying other LATEX macro packages) and TEX made it possible to format one's Docbook document with the DSSSL style sheets and obtain high-quality printed output. Armed with these free tools one could author, format and print Docbook documents at a time when SGML tools generally were scarce and expensive. With the advent of XML and XSLT more free tools have become available. The Docbook DTD is now available for XML, and the stylesheets have been rewritten in XSLT.

These combined features have made the Docbook DTD the DTD of choice for technical literature. The Linux Documentation Project is one well-known project that switched over from a private DTD to the Docbook DTD. Due to this strong position, the toolset for working with Docbook documents is growing rapidly, see e.g. `http://www.miwie.org/docbookinfo.html`.

How did it start and where is it now?

During EuroTEX 2001 in Kerkrade I had become interested in using ConTEXt because of the beautiful presentation styles used by Hans Hagen and several other speakers. While I was following the ConTEXt email list, I also became interested in ConTEXt's XML capabilities. These seemed so wonderful to me, that I *had* to understand how this could be done using TEX macro programming. I started asking questions. Sometimes Hans answers such questions with the suggestion that one take up some or other project. So he suggested that I start an XML mapping for Docbook.

I really had other plans, but I was so intrigued with ConTEXt's XML capabilities that I could not resist and gave it a start. As an added benefit, I would become more familiar with the Docbook DTD. When I started I certainly was aware that this would not be a small task. Docbook is such a large DTD, allowing its authors to use the hundreds of elements in innumerable combinations. But only while the project evolved did it become evident to me how large it really is.

Michael Wiedmann, who is interested in all possible tools to render Docbook documents, heard about the project soon after I started it. He made several contributions. His support and interest helped me to continue through the difficult phase when a project is no longer new, but you do not yet have anything really usable and you know all too well how much work still has to be done.

Now, a year later, I have some sort of an answer as to how it is possible to program ConTEXt's XML capabilities in TEX macros: Theoretically TEX macro programming is complete in the sense of having the expressive power of a Turing machine. Hans Hagen is one of the few programmers who can turn this theory into practice.

I also have a working XML mapping for DocBook documents in ConTEXt, which I call Docbook In ConTEXt (DIC). It contains good layout instructions for a number of often-used elements in their more common combinations.

Running Docbook In ConTEXt

Before one can typeset an XML file `myfile.xml`, one should create a TEX driver file `myfile.tex`, which should look something like this:

```
\input xtag-docbook

\starttext
\processXMLfilegrouped{\jobname.xml}
\stoptext
```

Then TEX is invoked as: `texexec myfile.tex` to get a dvi file, or as `texexec --pdf myfile.tex` to get a pdf file.

In the driver file `xtag-docbook` is the file name of the module. The XML document is input with the `\processXMLfilegrouped` command. The filename `\jobname.xml` is always correct provided the driver file and the XML file have the same base name.

Alternatively, one can always use the same driver file, in which the name of the XML file is changed each time.

The ConTEXt documentation indicates that one can also run the XML file as

```
texexec --xmlfilter=docbook testxml.xml
```

This will not work because the name of the Docbook In ConTEXt module does not conform to ConTEXt's nam-

1. `http://www.oasis-open.org/docbook/`
2. `http://sourceforge.net/projects/docbook`

ing conventions. It works if the module is renamed as `xtag-doc.tex`.

Customizing Docbook In ConTEXt

A Docbook XML document is a normal ConTEXt document. The commands that make up a ConTEXt document are also at work when a Docbook XML document is processed. They are just one layer away from what the user sees. Therefore the output can be customized with ConTEXt's setup commands as for any ConTEXt document. The setup commands should be given *after* the Docbook In ConTEXt module has been read, so that they override the default setup commands in the module. If you do not give additional setup commands, ConTEXt's defaults are applied. This is an example of a driver file with ConTEXt setup commands:

```
\input xtag-docbook
\setupindenting[medium]
\setupheadertexts[section][pagenumber]
\setupheader[leftwidth=.7\hsize,
             style=slanted]
\setuppagenumbering[location=]
\setupitemize[each][packed]
    [before=,after=,indentnext=no]

\starttext
\processXMLfilegrouped{\jobname.xml}
\stoptext
```

Docbook In ConTEXt also defines a number of setup commands and other customizations of its own. We describe a few of them in the following subsections.

Section blocks ConTEXt always applies pagebreaks around section blocks, and it treats the Table of Contents and the Index as chapters. This behaviour can be changed with the `pagebreaks` option of the `\setupXMLDB` command:

- `\setupXMLDB[pagebreaks=all]`: Default ConTEXt behaviour.
- `\setupXMLDB[pagebreaks=sectionblocks]`: ToC and Index do not start a new page, and they are treated as sections. All other section blocks retain their default ConTEXt behaviour.
- `\setupXMLDB[pagebreaks=none]`: In addition to the `sectionblocks` option, bodymatter, appendices and backmatter do not start a new page.

Titles Titles are formatted with a command of the form `\XMLDB` *element* `title`, where *element* should be replaced with the name of the element to which the title belongs, e.g. `\XMLDBarticletitle`. These commands can be redefined. They take one argument, the title. For example, the article title could be redefined as:

```
\def\XMLDBarticletitle#1%
   {\startalignment[left]
    \bfb #1
    \stopalignment
    \blank}
```

Note, however, that in Docbook documents the article title is often placed in the `articleinfo` part. In that case one should redefine `\XMLDBarticleinfotitle`. Similarly for book and chapter titles. Section, subsection, etc., titles are mapped to ConTEXt's `\section`, `\subsection`, etc., commands. Therefore they can be customized with ConTEXt's usual `\setuphead` command.

blockquote, epigraph and attribution The Docbook elements `epigraph` and `blockquote` have their own setup commands `\setupepigraph` and `\setupblockquote`, which have the following options:

- `narrower`. Both `epigraph` and `blockquote` are formatted using ConTEXt's narrower environment. The value of this option is a list of `left`, `right` and `middle` that is passed on to the `\startnarrower` command. See the ConTEXt documentation for `\startnarrower` for the effect of these settings.
- `quote`. The value is `on` or `off`. When on, quotation marks are applied as with ConTEXt's quotation environment.
- `command`. The value is a command or set of commands, which are applied at the start of the `narrower` environment.

The element `attribution` is customized with the command `\setupattribution`, which has one option: `command`. The value is applied at the start of the attribution.

More customizations Customization has only recently obtained the attention it deserves. By now more setup commands like those for `blockquote` and `epigraph` have been added, and others will follow. The distribution contains a document `Customization.xml` which will contain an up-to-date description of the customization options.

Example of customization Fig. 3 demonstrates the effect of customization. It shows again the article of fig. 1, but this time the driver file contains some extra customization commands.

```
\setuphead[section]
  [style=bia,number=no,align=left]
\setupepigraph
  [narrower={1*right},command=\bi]
\setupattribution[command=---]
\setupXMLDBlists[notoc]
\setupXMLDB[background=off]
```

DocBook In ConTeXt

Simon Pepping
Michael Wiedmann

Installation

There was things which he stretched, but mainly he told the truth.
—Mark Twain, Huckleberry Finn (1884)

Change directory to the top directory of one of the `texmf` trees of your TeX installation, e.g. `/usr/share/texmf`, and `untar` the distribution file `DocbookInContext.tar.gz`. Then run the command `mktexlsr` for that tree, e.g. `mktexlsr /usr/share/texmf`. Test the result by issuing the command `kpsewhich xtag-docbook.tex`. The reply should be the path of one of the files just installed, e.g. /usr/share/texmf/tex/context/DocbookInContext/xtag-docbook.tex.

Usage

Run context on a Docbook XML file using `xtag-docbook.tex` as the map file. Context offers several possibilities to do this, see the context XML manual `example.pdf`.

One option is to construct a TeX file that inputs the mapping file `xtag-docbook.tex`, and in the text block inputs the XML file with the command `\processXMLfile-grouped`. For example:

```
\input xtag-docbook
\setupheadertexts[section][pagenumber]
\setupheader[leftwidth=.7\hsize,style=slanted]
\setuppagenumbering[location=]
\setupitemize[each][packed][before=,after=,indentnext=no]

\starttext
\processXMLfilegrouped{\jobname.xml}
\stoptext
```

Fig. 3: The article of fig. 1 as formatted by ConTeXt with the Docbook In ConTeXt macros and the customization commands discussed in the text on page 393.

```
\def\XMLDBarticleinfotitle#1%
  {\startalignment[middle]%
   \bib #1%
   \stopalignment\blank[1*big]}
\defineXMLattributeaction
  [para][role][first]{\bf}
```

The first command is a normal ConTeXt command, changing the style of the section heads to bold italic, normal size, no number, and right-aligned.

The second command moves the right margin of the epigraph inward by 1 unit (the size of a unit is determined by ConTeXt's `\setupnarrower` command).

The third line does *not* set the font for the attribution to italic, which Docbook In ConTeXt's default setting does.

The fourth command switches the Table of Contents off. The Lists of Figures and Tables are switched

off by default.

The fifth command switches the background of program listings off.

The sixth command redefines the layout of the article title. More precisely it redefines the title in the `articleinfo` element, which is where the title usually lives. There is no setup command for redefining titles. A definition is required of a command that takes one argument, the text of the title.

The seventh command causes the first paragraph of each section to be printed in bold. The command defines an action when the `role` attribute of the `para` element has the value `first`. The action is `\bf`. The stylesheet takes care to put this action in a group. This action works because *this* XML document marks each first paragraph with the value `first` for the `role` attribute (see fig. 1).

Other tools for the same task

Docbook In ConTEXt is not the only tool for typesetting a Docbook document.

The Docbook XSLT stylesheets for printing The canonical tool for typesetting any XML file is XSLT+FO. An XSLT stylesheet is used to define the desired output in terms of Formatting Objects (FO). The FO description can be thought of as a formatter independent layout description. Then an FO processor is used to produce actual printed output, on paper or as an electronic document.

XSLT stylesheets for Docbook, written by Norman Walsh, have been available for several years. They implement a large part of the Docbook elements — implementing all elements seems almost impossible. And they are extensively parametrized, so that users can customize many aspects without modifying the XSLT code.

The objective of XSLT+FO is: one stylesheet, many processors. Several FO processors are available, among which are two free tools: FOP and TEX. FOP is a dedicated FO processor that produces output in PDF and a range of other formats. It is available from the Apache web site.[3]

TEX can be used as an FO processor using David Carlisle's XML parser `xmltex` and Sebastian Rahtz's `passivetex` package. `xmltex` works in much the same way as ConTEXt's XML processor. It allows one to register commands for an element, which will be applied when that element is started or ended. It does not depend on LATEX or ConTEXt.

`passivetex` does the same as Docbook In ConTEXt. It is a collection of commands to be called by `xmltex`. But it does not register commands for the elements of a specific DTD. Its commands apply to Formatting Objects. This is possible because Formatting Objects are written as elements in an XML file. In that way `passivetex` turns TEX into an FO processor.

Unfortunately, neither FOP nor `passivetex` do a good job with the Docbook XSLT stylesheets. The stylesheets make use of features which are not implemented in these FO processors.

The `db2context` project In March 2003 the first release of the `db2context`[4] project was announced. It is a sister project to the `dblatex` project, and shares some code with it.

The `db2context` project uses XSLT stylesheets not to produce FO, nor HTML, but to produce a ConTEXt file. This may at first seem an odd application of XSLT, and it certainly is not as intended. But it is used quite often by LATEX and ConTEXt users, with success. XSLT has good logic to deal with the problems of transforming an XML file. Writing a good transformation in XSLT is certainly easier than writing one in TEX macros.

Another advantage of this approach is that it allows one to customize the resulting ConTEXt file. In that way, what one cannot get out of the `db2context` style sheets, one can achieve by editing the ConTEXt file. This procedure is a major sin against XSLT orthodoxy, because all formatting should, and *can*, be specified in the XSLT stylesheet. But it is a great advantage for those who know how to achieve their desired style in ConTEXt and do not know much XSLT. In such a situation, who is stopped by orthodoxy?

I do not know the current state of the `db2context` stylesheets.

Future plans

Currently, Docbook In ConTEXt is not completely integrated with the ConTEXt distribution. I have strictly used the ConTEXt API wherever I could, and avoided developing my own variants. But I have preferred to develop this module in separation from the development of ConTEXt itself. The time has now come to work on a better integration. I hope this can be achieved over the next year.

If good, customizable XSL stylesheets for Docbook exist, and if ConTEXt could be an FO processor for the resulting output, then why would it be a good idea to spend so much effort on writing a special Docbook stylesheet for ConTEXt?

In the ConTEXt community the idea of a special Docbook stylesheet for ConTEXt has been greeted with enthusiasm. Apparently, here the theory of one stylesheet for many processors succumbs to the practice that users prefer to work with their tools of choice. For a popular set of tools like Docbook and ConTEXt users afford the effort of another style sheet. Such a style sheet

3. `http://xml.apache.org/fop`

4. `http://sourceforge.net/projects/dblatex`

is more manageable for them and running the required tools is easier.

On the other hand, until now, *I* have spent most of the required effort. And *my* answer tends to be: Maybe it is *not* the best way to support Docbook and XML in ConTEXt. Maybe it would be more useful to work on FO mappings in ConTEXt.

Over the past year I have set up this stylesheet. I have investigated the main structure of Docbook and come up with a way to map that to a ConTEXt document. I have implemented a framework for the mapping. I have enjoyed doing all that, and my insight and skills in TEX macro programming have increased immensely. But the time has come that others take this over, add mappings for more elements, add customizations, add new ideas. I plan to move forward to more generic work to support formatting of XML documents using TEX as the typesetting tool.

Availability

Currently, Docbook In ConTEXt is available separately from the ConTEXt distribution, from my web site.[5] Michael Wiedmann's web page[6] with Docbook tools has a link to the Docbook In ConTEXt files.

Programming Docbook In ConTEXt

ConTEXt and XML ConTEXt can take XML documents as input. For that purpose it contains a non-validating XML parser, which recognizes XML tags as markup instructions. And it has an API (Application Programmer's Interface) which allows one to define actions for those tags. This is called mapping XML tags to ConTEXt. A typical mapping instruction is

```
\defineXMLenvironment[element]
   {start action}
   {stop action}.
```

During the start and stop actions one has access to the attribute values of the element. For example, this is how one reads the `align` attribute of an `entry` element (in a table) and issues the corresponding setup command for ConTEXt's `TABLE` environment:

```
\doifXMLvar{entry}{align}%
   {\expanded{\setupTABLE[align=
      \XMLvar{entry}{align}{}]}}
```

ConTEXt's programming interface for XML mapping is robust. Rarely if ever does one get tangled in expansion problems. But, as is seen in the above example, timing the expansion remains an issue: The command to retrieve the attribute value,

```
\XMLvar{entry}{align}{}
```

must be expanded before the setup command can be read by TEX. That is what `\expanded` does.

It is easy, is it not? In principle, writing a mapping for an XML document in ConTEXt is simple. You state which ConTEXt commands you want to use for the start and stop of each element, and ConTEXt takes care of the rest. Practice is more complicated, certainly if you want to write a useful, extensible and customizable mapping for a complicated DTD. In the following sections I discuss a number of noteworthy features of the Docbook In ConTEXt mapping.

Encoding and language

An XML document declares its encoding in the xml declaration at the start of the document. ConTEXt supports several encodings, among them the XML default encoding `utf-8`. Correctly reading an encoding is one thing. Making all characters available that can be addressed by an encoding is quite another thing. Unicode and its utf-8 encoding have brought all characters in the Unicode range, currently more than 50,000, into scope within a single document. At the moment many of these are mapped to 'unknown character'. Work is ongoing to bring more characters within reach of ConTEXt in a single document.

A Docbook document may declare its language in the `xml:lang` attribute of the document element. The Docbook in ConTEXt module contains at the moment translated strings for four languages: English, German, Dutch and Italian. These are used for automatically generated strings, such as the titles of the table of contents, the abstract, and the index.

Features for each element

Context stack Because an XML document has a tree structure, each element in the document has a list of ancestors. I call that the context, or the context stack, which contains the ancestors from the document root to the current element.

An element may push itself onto the context stack when it starts, and pop itself when it finishes. In principle all elements should do so. In practice a number of elements omit this because they or their children do not use the context stack in their formatting.

During formatting, the context stack can be inspected with the following commands:

- `\XMLDBcurrentelement`: The current element's name.
- `\XMLancestor#1`: The name of the ancestor at level `#1`. The current element is at level `0`.
- `\XMLparent`: The name of the current element's parent.

5. `http://www.leverkruid.nl/context`
6. `http://www.miwie.org/db-context/index.html`

- \the\XMLdepth: The depth of the context stack.
- \doifXMLdepth#1: Execute the following instruction if the context stack has a certain depth.
- \XMLDBprintcontext: Print the context stack in the log file (mainly for debugging purposes).

ConTEXt also defines the context stack. I have redefined it because ConTEXt's implementation did not satisfy my plans. Later I simplified my usage of the context stack. ConTEXt's implementation may now be perfectly satisfactory, but I have not checked this.

ConTEXt itself defines \currentXMLelement to hold the name of the current element. But it is only guaranteed to be valid while ConTEXt reads the XML tag. Indeed, the mapping of some start tags in Docbook in ConTEXt emit an \egroup command, which invalidates the value of \currentXMLelement.

Ignorable white space XML has the interesting feature of ignorable white space. It can be used to give the raw XML document a nice formatting and make it fairly readable. (It did not exist in SGML. As a consequence, SGML documents may be practically unreadable in an ASCII editor.) For applications that read the DTD, this feature is rather clear: white space in elements whose content may only consist of elements, is ignorable. For example, when the content model of a section only contains paragraphs, all white space that surrounds the paragraphs is ignorable. Applications like ConTEXt that do not read the DTD must resort to other means to find out whether white space is ignorable or not.

I have introduced a feature that is similar to the mechanism used in XSLT. One can declare that an element preserves white space with the command

```
\defineXMLDBpreservespace#1
```

and that it ignores white space with the command

```
\defineXMLDBstripspace#1
```

For these declarations to work, the elements should be on the context stack, and they and their children should use the command \XMLDBdospaces as the last command in their start and end tags. \XMLDBdospaces has the effect of ignoring spaces following the XML tag if the current element has been declared to ignore spaces.

In practice this is only used by elements that would suffer if white space is not ignored. Note that TEX itself already ignores a lot of white space, viz. all white space that it reads in vertical mode. In the example of white space surrounding paragraphs in a section, TEX would do the right thing by itself.

The correct functioning of \XMLDBdospaces is rather subtle. The following is a generic element mapping:

```
\defineXMLenvironment[xxx]
   {\XMLDBpushelement{\currentXMLelement}
    \XMLDBdospaces}
   {\XMLDBpopelement \XMLDBdospaces}
```

The command \XMLDBdospaces in the start tag is executed while xxx is the current element. So it ignores white space if xxx has been declared to contain ignorable white space. But the same command in the end tag is executed *after* xxx has popped itself from the context stack. So its parent is the current element, and the command ignores white space if that *parent* has been declared to ignore white space. That is indeed exactly what we want, because the spaces following the end tag </xxx> are in the parent's content.

There is a class of ignorable white space that TEX refuses to ignore: blank lines are converted to \par commands by TEX's input scanner, before we can tell TEX whether white space is ignorable or not. Even this does not always matter to TEX, because TEX discards empty paragraphs or paragraphs that consist of white space only. In the above example we could insert blank lines between the paragraphs without ill effect. But a blank line between the start tag of a footnote and its first paragraph has a notably bad effect: it introduces a \par command between the footnote number and the start of the text, so that the footnote number is in a paragraph by itself.

Such harmful blank lines can only be removed by preprocessing of the XML document. I wrote a tool to do that. It is a SAX document handler written in Java, which removes all ignorable white space. I call it 'Normalizer', and it is available on my web site.[7]

The output of this tool is not only good for the ConTEXt mapping. Looking over it is informative for authors of XML documents. Every amount of white space that is left by the tool, is regarded as meaningful white space by XML parsers. Is that really what the author wants?

Every element In principle every element should contain the following commands:

```
\defineXMLenvironment[xxx]
   {\XMLDBpushelement\currentXMLelement
    \XMLDBseparator \XMLDBdospaces}
   {\XMLDBpopelement \XMLDBdospaces}
```

That is, it pushes itself onto the context stack. It checks whether it should typeset a separator. And it checks whether it should ignore following white space. In its end tag, it pops itself from the context stack, and it checks whether its parent should ignore following white space.

The separator is used by such elements as author, which may generate a comma or the word 'and' between consecutive elements. By default it is set to \relax. A parent element should give it a suitable definition to be

7. http://www.leverkruid.nl/context

used by its children, and reset it to the default when it finishes.

Which element is next?

ConTEXt's XML parsing is event based. This means that the parser generates events, such as the start or stop of an element, and calls the associated actions. During the actions one only sees the current event. One cannot look back at past events, except for the data that one saved. One can certainly not look forward to check which elements follow. In contrast, XSLT is tree based. That means that one can scan all elements, preceding and following, in the formatting commands of an element. Event-based parsing may present serious problems to the programmer.

Is there a title? An abstract may but need not have a title. When there is no title, I want to print the default title 'Abstract'. Because of the event-based nature of the parse, one cannot at the start of the abstract look forward to see if a title will follow. One can only try to find a future event at which one may safely conclude that there is no title if one has not yet seen a title.

In an abstract the optional title may be followed by three element types: `para`, `simpara` and `formalpara`. When any of these elements is started, one may safely conclude that either the title has been seen or there is no title.

One solution is to save the title, and to redefine the mappings of each of these three elements, such that they output the title or the default title if there was no title. And then restore their default definitions for the following elements.

Another way to tackle this problem is to save the whole abstract and process it twice. In the first pass we check whether there is a title. During this pass, all output should be suppressed. In the second pass we first output the title or the default title if no title was found in the first pass, and then we output the content. Again this requires a redefinition of the three possible elements that may follow the title, so that they suppress their output at the first pass.

The third option is provided by TEX itself, not by the XML mapping. We redirect the typesetting of the abstract into a `vbox`. At the same time we save the title in the variable `\XMLDBtitletext`, which removes it from the typeset content in the `vbox`. Then we output the saved title or the default title if there is no saved title, and next we output the `vbox`. This is the best option, and I use it.

When I applied this method to other elements with the same problem, e.g. `note`, I noted that the `vbox` disturbs the line spacing. This lessened my satisfaction with this solution.

Just before I finished this article, I realized that the above solutions correspond to XML thinking, in terms of nodes that have or have not yet been seen. A TEX programmer, however, has another option. We can save the content of the abstract, and then check whether the markup string for the title, viz. `<title`, appears in it. I have now implemented this solution. ConTEXt has good string comparison macros. Nevertheless I have written my own solution, because I wanted to be absolutely sure that there is no expansion of the abstract text. I have had my share of expansion problems with accented and other non-ASCII characters.

In a section this solution would be a bit more problematic: we run the risk of saving a large chunk of text. Working with options one or two would not be fun, because there are more elements to be redefined. I think the only viable alternative would be to work with `\everypar`, because `\everypar` is TEX's low-level signal that there is new text. Fortunately, in a section a title is required, so I did not (yet) have to work out this problem.

This is an example of the problems that arise because in an event-based parse it is hard to determine if an optional element is not present. The following section presents an example of the problems that arise because in an event-based parse it is equally hard to determine when a certain group of elements is finished.

Sectioning Like many systems, ConTEXt partitions its document in frontmatter, bodymatter, appendices and backmatter (called section blocks). The section block governs such properties as the numbering of the chapters and sections. I use the end of the frontmatter to print the table of contents.

Docbook does not have the equivalent of section blocks. There is no single element that contains the frontmatter, the bodymatter or the backmatter. Therefore I analysed the top-level structure of a Docbook document, and divided the elements that may occur as top-level elements into frontmatter elements, bodymatter elements and backmatter elements. When the first top-level bodymatter element is seen, the frontmatter is complete and the bodymatter starts. Similarly for the backmatter.

For a book in Docbook the situation is rather clear: The bodymatter starts with the first `part`, `chapter`, `article` or `reference`. For an article the situation is much fuzzier. While I counted only 6 top-level frontmatter elements, I identified 56 top-level bodymatter elements.

The transitions between the other section blocks are fortunately more clearly marked. The complete analysis is contained in the documentation for the module itself.

The situation is programmed using the commands

`\XMLDBmayensurebodymatter`

and

`\XMLDBmayensurebackmatter`

All top-level bodymatter and backmatter elements execute the appropriate command. These commands check if the element is a direct child of the document element, i.e. if the depth of the context stack equals `2`, and if the corresponding section block has not yet been started. The current section block is kept in the variable `\XMLDBsectionblock`.

Note that this means that TEX grouping runs across the XML tree structure. The start of a node may close a section block, i.e., it closes a TEX group.

Specific elements

Tables Docbook uses the CALS table model. The ConTEXt format uses two different table models. One is the tabulate environment, which is based upon TEX's `\halign`. It is quite sensitive to expansion timing errors. The other is the TABLE environment, also called natural tables. It is a very powerful and flexible environment, with many customization possibilities using `\setupTABLE` commands. A special feature of this table model is that rows, columns and cells can be configured both before and after their content has been given, at any time before the end-of-table (`\eTABLE`) command.

Because ConTEXt's natural tables have many similarities to CALS tables, the mapping is in principle very easy: a `row` corresponds to TR, an `entry` to TD, `colspec` elements can be mapped to `\setupTABLE` commands.

There are three main complications.

- The top, bottom, left and right frames of a CALS table are determined by the `frame` attribute of the table; the `rowsep` and `colsep` attributes of the corresponding rows and cells should be ignored.
- CALS tables can have multiple `tgroup` elements, each with their own number of columns, and their own alignment and frame settings (`colspec` elements).
- Each `tgroup` may have its own `thead` and `tfoot` elements, which may contain their own `colspec` elements.

These requirements have resulted in the following model: The `table` element generates a ConTEXt table, i.e. the table float, using the `\placetable` command. Each `tgroup` element generates its own TABLE environment, i.e. the actual table.

The table is not opened by the start tag of the `table`, because at that moment the title is not yet known. Instead, it is opened by the start tag of the first `tgroup` (command `\XMLDBopentable`, which contains the ConTEXt command `\placetable`). The start tag of each following `tgroup` typesets the previous `tgroup` (command `\XMLDBendTABLE`). Before typesetting, the left and right frames are set up. The start tag of the second `tgroup` also sets up the top frame. The end tag of the `table` does the same as the start tag of the next `tgroup` would do. In addition it sets up the bottom frame of the table, and closes the `vbox` of the `\placetable` command.

The rest is careful attribute processing, and issuing the required `\setupTABLE` commands at the right time. Attribute processing generates a lot of overhead, because both the attribute names and their possible values have to be translated from CALS to `\setupTABLE`. That makes the code somewhat less readable, but the logic is quite straightforward.

Issuing the required `\setupTABLE` commands is a precise work.

- The start tag of the first `tgroup` applies the `frame`, `colsep` and `rowsep` attributes of the Docbook table (`\XMLDBopentable`), so that they apply to all TABLEs in this CALS table. The start tag of each `tgroup` applies its own `align`, `colsep` and `rowsep` attributes, within its own TABLE environment.
- `colspec` elements of a `tgroup` apply their attributes to the whole column of this TABLE. The `colspec` elements in the `thead` and `tfoot` elements, on the other hand, must save their attributes (`\XMLDBsavecolspec`); they will be applied per `entry` in the `thead` and `tfoot`.
- `row` elements apply their attributes immediately to the whole row.
- `entry` elements first check whether they are in a `thead` or `tfoot`; if so, they apply the saved `colspec` attributes. Then they apply their own attributes. This order is important. ConTEXt gives precedence to properties set up per cell over properties set up for the whole table or per row or column. But in this case we apply what was originally a column specification per cell, so we must take care of the precedence ourselves.

Revision history The revision history contains a number of revisions. Each revision specifies one or more fields out of five possible fields. I wanted to represent this in a table which should only contain those columns for which at least one revision specifies data. Programming this was my first challenge in this project.

Hans Hagen suggested the solution. The revision history is saved, and then processed twice.

For the first pass of the saved revision history, we define the revision fields such that they register themselves when they occur, but suppress all output. We also count the number of revisions, so that we will know which row must contain the bottom rule of the table.

Now we know which fields occur and we can set up the table and output its header row.

For the second pass we define the revision element such that it outputs the row with the fields. So that the fields are output in the same order as in the header row, regardless of their order in the XML document, we first save the fields in a revision, and at the end tag of the revision we output the whole row in the desired order.

I worked this out both in ConTEXt's tabulate environment and with its natural tables. I decided to keep the solution with the natural tables, because natural tables are more flexible and less prone to expansion errors.

This procedure demonstrates a powerful feature of ConTEXt's XML processing: It is possible to save a node of the XML document with its subtree; in other words, the content of an element, complete with embedded elements, is saved in a variable without parsing. Later one can process the saved subtree as often as one likes. In between one is free to redefine the behaviour of the embedded elements. In TEX's macro language this is quite normal behaviour:

```
\def\savevar#1{\def\var{#1}}
... % redefine \processvar
\processvar{\var}
... % redefine \processvar
\processvar{\var}
```

In other programming languages it is not nearly as easy. Saving a node with its subtree in a SAX content handler so that it can be processed later is not a trivial task.

It is a disadvantage of the above procedure that the code is not easily read, certainly not if one is not used to the procedure. Recently, I have discussed an alternative procedure using Giuseppe Bilotta's `xdesc` module. It would achieve the same result but make the programming more transparent. Another advantage would be that it is more easily customizable by the user.

Program listing and CDATA I have spent an enormous amount of time on program listings. At first it seemed easy: ConTEXt has a verbatim environment which suits our purpose.

Then it was pointed out to me that some program listings contain `CDATA` sections, which were not treated well by my solution. I realized that a program listing is not really a verbatim environment because it does not disable XML tags. I dived deep into ConTEXt's verbatim environment and came up with a variant that supported two types of verbatim: one real verbatim for `CDATA` sections and one that did only line oriented layout for program listings. Moreover, it was nestable, so that it could deal with `CDATA` sections within program listings.

But it remained problematic to get it quite right. When the end of the `CDATA` section or of the program listing element was followed by text on the same line, this text was lost. And my white space tool did exactly that: put the following text right behind the end of the program listing element.

When I revisited the problem a few months later it dawned on me that the whole verbatim approach was wrong. Neither `CDATA` sections nor program listing environments have anything to do with TEX's notion of verbatim. `CDATA` sections just disable XML markup. They may occur anywhere in an XML document, and have no semantic meaning. Indeed, an XML parser does not even report whether `CDATA` sections are used in an XML document; it simply resolves them.

For the program listing I found a simple solution. It avoids scanning a whole line at a time, therefore it avoids scanning the text following the end of the program listing with the wrong catcodes in place. It uses `\obeylines` and `\obeyspaces` and it places struts at the start of a line to prevent the leading spaces to be discarded by TEX's paragraph mechanism. That is all, and it does the job well.

Hyperlinks, URLs and external documents Docbook documents mark hyperlinks with the `ulink` element; the url is contained in its `url` attribute. If we were writing HTML documents it would be easy:

```
<ulink url="URL">text</ulink>
```

would be translated to:

```
<a href="URL">text</href>
```

and the browser would do the rest.

But this does not suffice for PDF documents. Links to PDF documents should be treated differently from links to other documents, and relative links to non-PDF documents are not allowed. Therefore, we have to analyse the URL and complete it if necessary.

In ConTEXt strings can be split into parts with commands like

```
\beforesplitstring
    string\at substring\to\var
```

which splits `string` at `substring` and stores the first part in `\var`. I use this and similar commands to check whether the URL has an "authority" (this is the term used by RFC2396, which specifies URIs; usually it is called the protocol, e.g. `http`) and whether it is an absolute or a relative URL. If a local file is specified, we also check whether it has the extension `pdf`. Links to local PDF documents are created using the ConTEXt command `\useexternaldocument`, links to other documents use the ConTEXt command `\useURL`.

URLs like `slashdot.org` pose a special problem. Is it a web server, or a file in the current directory? Cf. the URL `myfile.html`, which has exactly the same pattern. After the terminology of RFC2396 I call these abbreviated URLs. By default they are not recognized.

Thus `myfile.html` is correctly linked to as a local document, while `slashdot.org` is incorrectly linked similarly. The user can switch recognition of abbreviated URLs on by setting `\XMLDBcheckabbrURItrue`, and can switch if off again with `\XMLDBcheckabbrURIfalse`.

Unfortunately, I do not know how to get the working directory in a ConTEXt run, so relative URLs are currently not properly completed.

Customization

For a long time I did not pay much attention to customization. Recently, I received requests to make a mapping for the `blockquote` and `epigraph` elements. Together with that request a discussion arose on the ConTEXt mailing list about customization. As a consequence, these two elements and their child element `attribution` have proper setup options, namely, the commands `\setupblockquote`, `\setupepigraph` and `\setupattribution` (discussed on page 393). Since then several customization options have been added and surely more will follow.

The same discussion on the ConTEXt mailing list touched upon attributes whose range of values is not constrained. An example is the `role` attribute of any element. It is not possible to define actions for such attributes in the stylesheet, because the possible values are not known. The idea arose to put a hook in the stylesheet for the user's own formatting command, called attribute action. The user can define such an action as follows:

```
\defineXMLattributeaction
    [para][role][first]{\bf}
```

The stylesheet invokes the attribute action within a group in order to limit the user's actions, such as changing the font, to this element. Therefore ConTEXt cannot invoke the attribute action automatically; it cannot know where it should do so. For example, some mappings for the opening tag invoke `\egroup`; if the attribute action had been invoked automatically, its scope would be ended immediately.

The author of a Docbook document may provide a value for the `role` attribute for every element. But I am wary of enabling an attribute action for every element. It would put every element in a group, and I am not sure about the effects. For now attribute action has experimentally been implemented for `para` and `programlisting`.

I am not sure how far customization can go. Enabling extreme customizability would come down to defining a new language for describing the formatting of a Docbook document. This would go too far. On the other hand, customizability is a strong feature of ConTEXt. It is not difficult to add customizability options to the stylesheet; ConTEXt has some good commands for that.

Docbook in `xmltex`

Would it have been possible to write these stylesheets for Docbook based on `xmltex`? Of course.

Since `xmltex` is independent of LATEX or ConTEXt, it would in principle even be possible to write Docbook in ConTEXt in `xmltex`, that is, to use `xmltex` as the parser and base the commands on ConTEXt. But, as expected, there are conflicts when `xmltex` and ConTEXt are used together.

`xmltex` and LATEX would have been a good basis for a Docbook stylesheet. The approach would in principle be the same as that used in the current project. Like the ConTEXt XML parser, `xmltex` allows one to register actions for each element. The syntax regarding attribute specifications is a little different. Both allow one to specify an action for the start and one for the end of an element.

The major difference would be that between LATEX and ConTEXt, which is quite large. Three significant differences may be mentioned.

- XML support in ConTEXt is actively maintained and evolving. `xmltex` is not.
- Unicode and utf-8 support in LATEX is extensive, due to components of the `passivetex` package. In ConTEXt many Unicode symbols are yet undefined at the time of this writing. It would be an advantage if `passivetex`'s Unicode support would be ported to ConTEXt.
- ConTEXt has a strong integration of MetaPost and PDF. I have not made much use of this feature. For an example of what is possible, see the admonition symbols in Docbook In ConTEXt produced with MetaPost.

Acknowledgements

Michael Wiedmann contributed the earliest mappings for several elements, a.o. `mediaobject`, `table`, and `ulink`. He also contributed the implementation of string literal files, and the string literals for English and German. Giuseppe Bilotta contributed the string literals file for Italian, and Pablo Rodriguez contributed the same for Spanish. Richard Rascher-Friesenhausen made several contributions, a.o. the MetaPost admonition icons.

And of course, nothing of this would have been possible without Hans Hagen's ConTEXt. ConTEXt is the framework in which Docbook In ConTEXt runs and a rich source of examples of excellent TEX macro programming.