Linking Outside the Box

Bob Stayton, Sagehill Enterprises

Abstract

When managing large technical documentation sets, the ability to cross reference provides huge gains:

- Cross references let you document information once. When information is needed in another context, a writer can cross reference to it rather than repeat it.

- Writers can specialize. Writers don't need to become experts on every subject when they can refer the reader to the best information on a subject.

- Update maintenance is easier because the information can be updated in a single place rather than several places. Also, link text can be automatically generated.

- Translation costs are reduced because there is less content to translate.

XML supports cross referencing using the ID/IDREF mechanism, but such links can't go outside the current document. What if the needed information is in another document? Hard-coded URLs are notoriously difficult to maintain, and the XLink standard has not been widely implemented.

The DocBook community has implemented olinks, a powerful and versatile cross referencing system that lets you wire together a collection of documents with maintainable links. By specifying just two attributes, you can link anywhere in a collection of documents. The olink system saves cross reference information for all documents in a simple XML database. At runtime, the XSL stylesheet reads the database, forms the appropriate URI, and generates up-to-date hot link text.

Before olinks, authors could only provide a general reference to the title of another document. It was up to the reader to locate the other document and find the information in it. Now the reader can jump right to the specific reference in another document. Such cross references are far more likely to be used, and provide a richer experience for the reader.

The olink system has a full set of options:

- You can specify different styles for assembling the generated text, either globally or for individual references.

- You can cross reference between HTML and PDF documents.

- You can set up a language fallback sequence to find the closest language match for a link.

Olinks also enable DocBook to deliver on the promise of true modular content authoring. If olinks are used for internal as well as external linking, then file modules can be small and valid, with no unresolved IDREFs. Modules can be combined with XInclude into larger documents, with the stylesheet resolving all cross references from the database.

Table of Contents

Introduction 2
Background 3
Advantages of cross referencing 3
Designing a cross referencing system 4
Cross referencing in DocBook 5
What about XLink? 5
Resolve to rendered documents 7
Olink XSL implementation 7
Two new olink attributes 8
Target data format 9
Processing olinks 11
New DocBook capabilities 13
Modular XML files 13
Asynchronous processing 14
Links to open source doc 15
Faster XML web service 15
Language fallback 15
Conclusions 16

Introduction

Cross referencing is a powerful tool when writing technical documentation. It lets you document something once in an authoritative manner, and then refer to that information from other points in the documentation as needed. There is less maintenance because information is not repeated, and the reader always has access to the most accurate information. With modern hypertext readers such as HTML and PDF browsers, the reader can jump immediately to the references.

Yet the basic linking mechanism built into XML does not support cross referencing to targets outside the current document. This has forced authors to put all the content within the scope of cross referencing into a single XML document. This is an unwieldy solution that runs counter to the goals of reusable content modules.

The W3C XLink Recommendation was supposed to address this deficit for XML, but it does not provide the methods to process such links, and almost no implementations of XLink exist.

To satisfy this need, the DocBook community created the `<olink>` element. This elements lets the author identify an external document and a reference point within that document. The XSL stylesheet then processes that information into a link. This scheme enables several important new features in a DocBook publishing system:

Links between documents	Now you can make cross references to content in other DocBook documents. You no longer have to put all the content in one document to form cross references.
Modular content files.	Create smaller XML files that can be processed individually, merged into larger documents, and recombined into different documents.
Asynchronous processing.	Process different documents at different times while maintaining cross references between them.

Links to open source documentation.	Maintain cross references to a collection of open source documentation, which is often the most authoritative content on a given subject.
Fast XML web service.	Process on the fly only a small amount of requested content instead of an entire book or set.
Language fallback	If your translated document has cross references to a document that did not get translated, then the link should fall back to an available language.

Olink enables these features because it breaks free of the constraint of using ID/IDREF for cross references. Yet this system has been implemented without any changes to the XML standards.

Background

The cross referencing system that is built into XML is based on ID and IDREF attribute types. An ID attribute establishes a target name for an element, making it a potential destination. Then an IDREF attribute is used in a cross reference element to point to a target ID.

XML requires that the ID reside in the same document as the IDREF. That way the document validation step can confirm that cross references are valid. This requirement means that any content you want to cross reference to must be in the same document. In DocBook, this is typically done by using larger document containers such as `<book>` or `<set>` elements to hold everything you want to cross reference among.

Cross referencing within a large set using ID/IDREF creates many problems. It is unlikely you would put all the content in a single file, so writing and editing means working with system entity files or XIncludes that contain part of the content. But each such file cannot be validated if it has IDREF links to targets outside of itself. And when you want to process any of your content, you have to process all of it at once, even if you make a minor change in one component. These burdens in the editing and processing stages inhibit the use of cross references.

Advantages of cross referencing

Cross referencing provides huge gains for managing a large documentation set:

- Cross references allow you to document information once. When part or all of some information is needed in another context, there is no need to repeat the information; a writer can just cross reference to it.
- Writers can specialize. Writers don't need to become experts on every subject when they can just refer the reader to the best information available on that subject.
- Less translation. By pointing to content that is already translated, there is no need to translate the same information more than once.
- Easier update maintenance. Whenever the same information is repeated in more than one document, any updates require finding and updating all such documents. If the information appears once and the other documents only cross reference to it, then you only have to update the information in one place.
- Users get more accurate information. If the information appears in more than one place, there is a chance that only one will be updated because no one remembered to update the other instances. If the information is structured as one instance and several cross references to it, then the user will always get the updated information.

- Direct cross references are more useful than generic textual references. Without the machinery to resolve cross references, an author has to resort to general cross references to a document's title. That leaves it up to the reader to locate the other document and find the information within it. A cross reference that takes the reader directly to the new information is more likely to be used.

Cross references become even more useful when the processing machinery can automatically generate their cross reference text from information in the target of the cross reference (e.g., the title and chapter number in the target document).

- Less maintenance. If a section title is updated, no one has to remember to also update the text for every cross reference which refers to that title. Likewise, if a numbered target (e.g., Figure 3-2) is moved so that its number changes, the cross reference number is automatically updated when next processed.
- Users get more accurate information. The hot link text always accurately reflects the title and number of the target that they land on.
- Less translation. Empty `<xref>` elements do not need to be translated. The localized hot text is automatically picked up from the translated target document.
- Flexible presentation. While an HTML stylesheet can make the cross reference into a hot link, a print stylesheet can rearrange the words and possible add a document title and page number.

These benefits of cross referencing come at a cost: that of managing the cross references. The standard ID/IDREF mechanism manages cross references quite effectively by validating them when the document is validated. Cross references implemented without ID/IDREF must be managed carefully if you don't want to degrade your documents. If a reader follows a link and lands in the wrong place (or no place), then they are less likely to trust the information they are reading. The cross referencing machinery must help maintain the accuracy of cross references, and it must be easy and automatic. This is particularly true when creating and maintaining large documentation sets. There are never enough eyes to check every cross reference against its target for each product release cycle or publication.

Designing a cross referencing system

A full-featured cross referencing system should be able to do the following:

- Cross reference within large documentation sets without having to process them all at once as a single document.
- Cross reference to content that cannot be in the current document because it resides in a different department or is processed on a different schedule.
- Cross reference to open source documents you are reprocessing for release on your own media.
- Cross reference to open source documents that are already processed and released.
- Automatically generate text for all of these cross references, with possibly different text for different output formats.
- Flag unresolved cross references.
- Publish a collection of target data through which you want your own documents to be crossreferenced.

Cross referencing in DocBook

The cross referencing system described above has been implemented in DocBook. There are four cross reference elements in DocBook, three of which cannot be used in such a system.

- In a `<link>` element, the content of the tag becomes the displayed hot text of the link. It uses a `linkend` attribute, which is of type IDREF. The value of the attribute must match some attribute of type ID somewhere in the document. In DocBook, these are generally the common `id` attribute on some element. It is a validation error if no match is found within the document.

- An `<xref>` element differs from a `<link>` element in that `<xref>` is empty and expects the processing stylesheet to automatically fill in the hot text using information from the target of the link, such as its title. But it too uses a `linkend` attribute to establish an ID/IDREF link within the document. Again, it is a validation error if no match is found within the document.

- A `<ulink>` element is a URL link. It uses a `url` attribute to establish a direct link to a URL that a browser is expected to be able to follow. Generally the element's content is used as its hotlink text. But if the element is empty, the value of the `url` attribute is used as the hot text. Although a `<ulink>` can cross reference to another document, the link is hard wired and cannot generate text.

- The `<olink>` element is designed for linking to other documents. It does not use ID/IDREF attributes, so there is no requirement that the target of the cross reference be in the same document. The processing stylesheet is expected to resolve the entity and establish the link to the other document in a suitable fashion.

What about XLink?

XLink was designed to establish links between XML documents. XLink received the status of a full W3C Recommendation on 27 June 2001. See XML Linking Language (XLink) Version 1.0 [http://www.w3.org/TR/xlink/]

XLink uses the idea of global attributes in a separate `xlink` namespace to make elements in a DTD into XLink elements. Thus DocBook's `olink` could be made into an Xlink element by adding new attributes to the DocBook DTD. The full XLink spec provides for multiple links and bidirectional links between local and remote resources. It also lets you maintain link information outside the documents themselves in separate link bases. It was expected that many applications would arise to make use of these extended link features.

But XLink also supports simple XLinks, which is the type that document authors are currently used to. Simple XLinks point from the current element to another element, which could be in the same document or another document. A simple XLink can have other XLink attributes such as role, title, and actuate to modify its behavior. But it is always a single link from here to there.

It is the XLink `href` attribute that establishes the link. It looks similar to an HTML href attribute. The first part is a URI that points to a document. It could be a URL such as those used in HTML documents, or it could be a more general URN (Uniform Resource Name) with some mechanism for resolving the name to an actual resource. A second optional part, known as a fragment identifier, follows a # character and points to some location within that document. XLink permits fragment identifiers to use the powerful syntax of XPointers to locate information within the document. But it also supports the simplest form, which is just the value of an ID attribute within the target document. Thus our earlier olink example could be expressed as an XLink as follows:

```
<xref XLink:href="../OS-Userguide/book.xml#Mousing"/>
```

This assumes that a new `XLink:href` attribute has been added to the xref element in the DTD.

As you can see, this direct syntax is considerably simpler than the indirect references that olink had to resort to for making links between XML documents. This syntax establishes an unambiguous logical link between the current document and a point in the other document.

But how is this XLink to be interpreted when the current document is processed into HTML? Since this is an `xref` element, the stylesheet is expected to generate the hot link text, but what should it say? Also, exactly what does the user see when they click on the link? It is unlikely the author intended the reader to be dumped into the middle of an XML document to display raw XML markup like this:

```
<section id="Mousing">
  <title>Using a Mouse</title>
  <para>This is how you use a mouse</para>
</section>
```

Both the generated text and the destination XML must be styled appropriately for the presentation of the current document. That's easy, just apply a stylesheet, right?

Well, it turns out that it isn't all that easy. In a W3C Note XML Linking and Style [http://www.w3.org/TR/xml-link-style/], Norman Walsh outlines many of the complications that can arise, even with simple XLinks. Here is a sampling:

- Should the other XML document be styled with the current stylesheet (assuming it applies), or the other document's stylesheet?
- If the latter, and the other XML document doesn't have a stylesheet processing instruction in it, how does one determine the stylesheet for it?
- If the generated hot link text refers to a numbered item in the target document, then the entire target document must be styled to get the correct number for the reference. Simply counting XML elements in document order won't work because a stylesheet can transform the elements into a new order.
- XLinks that embed content from another resource can alter number sequences in complex ways.

Several more complications could be added to the list:

- How do you handle profiling (conditional text) that can generate different output from the same XML document? Profiling can affect number sequences, and whether the target is even available to be linked to. An XLink trying to style the document would need to know the precise means by which the document was profiled, which could be quite complex based on prefilter steps or specific attribute values.
- Parameters passed to the target's stylesheet can alter how the target document is styled. An XLink trying to style the document would need to know the parameters that affect processing.
- How do you handle chunking rather than displaying the entire document?
- How will the target document's own cross references be handled? When the target is styled for HTML, it will include HREFs to other HTML files. If the target document is styled on the fly, those HTML files may not actually exist, and those links will fail.

Resolve to rendered documents

The basic problem with XLink is how to resolve it to a styled result in your file. It appears you have to know their stylesheet and their profiling. You can't guess, you have to know with certainty what the results will be for your XLink to work.

If that is the case, then it becomes very difficult to link to the XML source document, which has several profiles in it and can be processed with several different stylesheets. Until that problem is solved, it is safer to cross reference to a rendered document with one profile and one style already applied. Cross references ultimately must land on a rendered version of the target if they are to be consumable.

But if we must link to a rendered document, why not just use `<ulink>`? Because ulink addresses are not flexible or robust enough. It is important to distinguish between the authoring process and the rendering process. You want your author to establish a logical link from one XML source document to another. You want a link that remains valid through several iterations of the product, and through several different renderings for print or online. A ulink that is valid in the current version may not be valid in the next. For ease of maintenance, the links should not require an editing pass through the document to update them for each new release. Rather, the processing machinery should resolve the abstract logical link to a rendered link for each release. An XLink or olink is the abstract logical link from XML to XML. An HTML href is a concrete working link.

At some point you render your document, that is, convert the XML to a final form for consumption. This conversion is done either ahead of time for packaging on the media, or on the fly in the server or browser. You want that logical link turned into a working link, using the actual rendered text, label, and URL of the rendered target. As was described above, it currently must resolve to a rendered version of the target document.

Olink XSL implementation

If cross references must finally resolve to a rendered document, then it is not necessary to process every target document just to resolve a link. If the appropriate information for linking was saved when the target document was rendered, then we can just use that information in the current document to form our links, and the target document would not have to be parsed. The datafile associated with olink in the DocBook stylesheets provides such a mechanism.

A separate stylesheet (`targets.xsl`) generates a structured distillation of information about all the potential cross reference targets in a document. The target data is stored in a separate file from the document. Once the target data is separated from the document itself, it becomes accessible for cross referencing applications.

The target data file is an XML document that consists of a set of nested `<div>` elements. Each div element records the information for a single strucutural element in the document, such as book, chapter, section, etc. Other potential cross reference targets that are not part of the structural hierarchy are recorded in `<obj>` elements.

The implementation of olink in the DocBook XSL stylesheets has the following goals:

- Make it easy for authors to link between documents.

  An author should only have to establish the logical link from one document to another. That means pointing to the target document, and to the specific target element within that document. The author should not have to worry about file locations, URL prefixes, or adding entity declarations to the DTD.

- Minimize overhead in maintaining links.

Over the lifetime of a document, much can change in how a link is resolved during the processing for each release. To avoid having to perform an editing pass through your document just to update links, the olink element should record the minimum information to establish the link, and not provide details for resolving or styling the link.

- Support HTML, print, and other potential output formats.

  To maintain the separation of content and formatting, olinks should be useful in all output formats.

- Provide flexibility in rendering links.

  Each organization may have its own style for rendering links. Also, external links may provide different information or be styled differently from internal links.

- Validate the links during processing.

  The processing machinery should flag broken links so they can be fixed.

Two new olink attributes

To meet these goals, it was necessary to add two new attributes to olink's attribute list. These new attributes appeared first in the DocBook XML 4.2 DTD.

`targetdoc` String identifying the document containing the target of the cross reference.

`targetptr` The locator of the target element within the targetdoc, currently using its `id` attribute.

To form an olink, the author provides just two attribute values. For example:

```
<olink
   targetdoc="OS-Userguide"
   targetptr="UseMouse"/>
```

The author can optionally provide link text in the olink content. For an empty olink, the stylesheet is expected to generate the text for the link.

To identify the target document, a `targetdoc` attribute is used. The `targetdoc` attribute is a simple identifier string (of type CDATA) that is resolved to an actual document by the stylesheet.

To identify the target element, a `targetptr` attribute is used. Its value must match that of an `id` attribute in the target document. Together, the two attributes make it easy to form an olink: you just have to identify the target document and the element within that document.

No other information need be supplied in the XML markup When the XSL stylesheet encounters an olink with these attributes, it uses XSL templates to resolve the link. As you will see, an optional xrefstyle attribute can be added to provide hints to the stylesheet on how to format a given olink instance.

Of course, the stylesheet cannot resolve such links on its own, so it must be supplied with sufficient information to do so. In the XSL implementation, the stylesheet is given a parameter (in a customization layer or at runtime) that points to a database file that provides that information. The advantage of supplying it at runtime is that it can be different for different builds of the document. A new release can refer to a new version of the target database. An HTML build can refer to an HTML rendering of the target documents.

The target database is a structured XML document with its own DTD. The data on potential cross references in a document is extracted using a special template in the XSL stylesheets. Then the data for the individual

documents is merged into a single target database, and the resulting filename is passed to the DocBook stylesheet as a parameter.. The next section describes this data format.

Target data format

A target database contains information for all the target documents the stylesheet may need during a given build. The overall structure looks like this:

```
<?xml version="1.0" encoding="utf-8"?> ❶
<!DOCTYPE targetset SYSTEM "/tools/docbook-xsl-1.52.1/common/targetdatabase.dtd" [
<!ENTITY ugtargets SYSTEM "/doc/userguide/target.db"> ❷
<!ENTITY agtargets SYSTEM "/doc/adminguide/target.db">
<!ENTITY reftargets SYSTEM "/doc/man/target.db">
]>
<targetset> ❸
  <targetsetinfo> ❹
    Description of this target database document,
    which is for the examples in olink doc.
  </targetsetinfo>

  <!-- Site map for generating relative paths between documents -->
  <sitemap> ❺
    <dir name="documentation"> ❻
      <dir name="guides"> ❼
        <dir name="mailuser"> ❽
          <document targetdoc="MailUserGuide" ❾   baseuri="userguide.html"> ❿
            &ugtargets; (11)
          </document>
        </dir>
        <dir name="mailadmin">
          <document targetdoc="MailAdminGuide">
            &agtargets;
          </document>
        </dir>
      </dir>
      <dir name="reference">
        <dir name="mailref">
          <document targetdoc="MailReference">
            &reftargets;
          </document>
        </dir>
      </dir>
    </dir>
  </sitemap>
</targetset>
```

❶ Set the database encoding to `utf-8` for the database, regardless of what encoding your documents are written in. The individual data files are written out in `utf-8` so a database can have mixed languages and not have mixed encodings.

❷ Declare a system entity for each document target data file.

❸ Root element for the database is `targetset`.

❹ The `targetsetinfo` element is optional, and contains a description of the collection.

❺ The `sitemap` element contains the framework for the hierarchy of HTML output directories.

❻ Directory that contains all the HTML output directories.

❼ Directory that contains only other directories, not documents.

❽ Directory that contains one or more document output.

❾ The `document` element has the document identifier in its `targetdoc` attribute.

❿ For documents processed without chunking, the output filename must be provided in the `baseuri` attribute since that name is not generated by the document itself.

(11) The system entity reference pulls in the target data for this document.

The data for each `<document>` is in a separate data file that is pulled in as a system entity reference. That data is structured in a manner similar to its target document. For every hierarchical element (book, chapter, section, appendix, etc.), there is a corresponding `<div>` element to record its target data. The nesting of div elements captures the parent-child relationships between elements. The value of each div element's `id` attribute would be captured, except that generated IDs are not included because they are not stable target names. In addition to the hierarchical elements, the collection includes all formal elements (tables, figures, etc.), and any block and inline elements that have ID attributes. These are stored as `<obj>` elements in the data set.

Here is an example of the target data for one document:

```
<?xml version="1.0" ?>
<div element="chapter" href="#publish" number="1" targetptr="publish">
  <ttl>Publishing DocBook Documents</ttl>
  <xreftext>Chapter 1</xreftext>
  <obj element="table" href="xsl-processors" number="1.1" targetptr="xsl-processor
    <ttl>XSL Processors</ttl>
    <xreftext>Table 1.1</xreftext>
  </obj>
  <div element="section" href="#xsl-arch" number="" targetptr="xsl-arch">
    <ttl>DocBook XSL Architecture</ttl>
    <xreftext>the section called &#8220;DocBook XSL Architecture&#8221;</xreftext>
  </div>
</div>
```

In general, generated ID values are excluded from the data set because they are not reliable targets for cross referencing. Certain elements without ID may also be included, however. Those hierarchical elements without an ID attribute may provide some text or context for styling the link data in the cross reference. For example, a chapter might not have an ID attribute, but it still has a number and a title. When a cross reference is to a section that has an ID within that chapter, the link could be styled to say "see the section `Using a mouse' in Chapter 3: Peripherals".

Note

Objects without IDs could still be pointed to using the W3C XPointer syntax, which is based on XPath. However, XPath is best applied to the original target document, not an extracted subset of information. This scheme might be extended in the future to include XLink and XPointer.

The `<div>` and `<obj>` elements are generated using a special template included with DocBook XSL that extracts the target data from each document. The template is triggered with a stylesheet parameter named *`collect.xref.targets`*. See the DocBook XSL stylesheet documentation for more information on using the parameter.

The optional `baseuri` attribute is a prefix for hrefs generated for the target elements in this document. If an HTML document is output as a single HTML file, then the `baseuri` is that filename, and `href` attributes in the data file are of the form `#elementid`. The stylesheets combine the `baseuri` and `href` attribute values to form an olink. If the HTML document was chunked on output, then `baseuri` is usually blank. Then the `href` attributes in the data file are of the form `filename.html#elementid`.

The optional `uri` attribute can be used to precisely identify a rendered version of a document. It would allow you to keep several data sets for the same document rendered in different versions, profiles, or output formats. All would share the same `targetdoc`, since that is general name used by olinks find the document. But when you assemble your target database for a given production run, you can select one rendering of the document that you intend the olinks to resolve to. The naming scheme is user defined. Here is an example using a URN:

```
uri="urn:Caldera:OS-Userguide:3.2:en:html:caldera.ss3
```

This indicates the target data extracted from version 3.2 of Caldera's OS-Userguide, in the English version, in the HTML rendering, rendered by Caldera using its "ss3" stylesheet.

Processing olinks

The processing of olinks with the DocBook XSL stylesheets can be summarized as follows:

1. Before processing a document with olinks, generate target data files for each target document.
2. Create a single target database file that pulls in the individual data files using system entity references.
3. Pass the database filename as the *`target.database.document`* parameter to the XSL processor.
4. XSL looks up each olink reference in the database.
5. XSL styles the target data using the current stylesheet.

Each of these steps is described in detail below.

Generate data files

A target data file should be produced for each target document, and for each different rendering of a document. Such a rendering encompasses any profiling done from the original XML source file, as well as any transformations performed by the stylesheet. It is such renderings that are the target of rendered cross references from other documents.

In this scheme, the data file must closely match the rendering of the document. Effectively, the datafile should be produced as a side effect of styling the document, at the same time the document output is pro-

duced. But it should also be possible to extract the target data without producing the actual rendered output. The stylesheets support both modes of operation.

For a rendering to print, an additional step must be taken to extract the page numbers for each target after the document has been paginated. These page numbers are merged back into the data file based on the ID values associated with them. This feature is dependent on an extension to the FO processor.

During the development process, a target data file should be generated as often as the document is changed. The final processing run of a document before publication should produce a final data file that can be targeted by other documents.

Merge into target database

Each document with olinks has a cross referencing scope, which is the collection of documents it is targeting for cross references. To process the olinks in a given document, the target data for all documents in that scope is merged into a single target database. The application can manage this in several ways.

If you are building a stable collection of documents that you cross reference among, you may just merge all of them into one database. Even if document A does not reference document B, it might after further editing or in a future release. If you can clearly define one scope and manage it as one database, then the job is easy.

If you are building documents that refer outside your own collection, then you need to be more careful. Each document may have its own scope, which is defined by the collection of olink `targetdoc` values in the document. The processing application may assemble a custom database on the fly from a collection of individual target data files. It is your job to make sure the target data is available for any of those target documents.

It is not necessary to move a lot of data around to form such a database. A database is a merger of several XML documents. This can be accomplished today using a wrapper around system entity references or xincludes. On the other hand, some shops might use Makefiles to load target data into a relational database such as MySQL, and then extract a custom target database document for each build.

Within a target database file, each document's target data is wrapped in a `<document>` element. See the section called “Target data format” for a complete description. Additional attributes of `<document>` are filled in when the database is assembled.

The database also may have a top-level `sitemap` child whose structure describes an output hierarchy of directories. Each `dir` element in the sitemap has a directory name attribute. Directory elements can be nested, as on a filesystem. At runtime, the applicaton passes in a parameter indicating the location in this directory hierarchy where the current document will be placed. This permits the stylesheet to compute a relative HREF from each processed document to each target document. If a target document is not in the output hierarchy, then its data container must have a `baseuri` attribute that provides the first part of its HREFs.

Pass database reference

However the target database is assembled, it is passed at runtime to the DocBook stylesheet as a parameter. The application manages which database is used for each document. Put the pathname to the database document in the `target.database.document` parameter.

Look up olink data

As the document is processed with the stylesheet, the olink template has a routine to look up the data it needs from the database. It uses the `targetdoc` attribute in the olink to locate the document container it needs in the database. It is an error if it isn't there.

Then it searches for the `targetptr` attribute among the `<div>` and `<obj>` descendants of that document container.

Style each olink

Once the proper `<div>` or `<div>` element is found in the database, the olink template uses the data in that element to style the link. Depending on the attributes and content of a given olink, this may mean just generating the HREF, or it may include generating the cross reference text. In the case of a print document, the template may insert a page reference or add a document title.

A generated cross reference text string generally uses the already assembled string in the `<xreftext>` element that was added to the data file when the target document was processed. That string represents the style of cross references in the target document. Or an olink could use a customizable text template to style the raw data fields into a cross reference style that matches the current document. The generated text could use the same template as for `<xref>` elements, or it could use a template customized for olinks.

Currently olinks use the `xref` context for text templates to style links. Those templates are part of the common stylesheet templates that are shared between html and print output. If different styling of olinks is needed, a set of text templates using a new `olink` context rather than the `xref` context could be created.

An optional `xrefstyle` attribute can be added to a given `olink` element to provide hints to the stylesheet on how to format the link. The DTD does not specify standard values for the attribute, so stylesheets are free to implement whatever scheme they want. The DocBook XSL stylesheets provide three ways to use the `xrefstyle` attribute:

- If the attribute value begins with `template:` then the rest of the text after the colon is taken to be a gentext template to use for that reference.
- If the attribute value begins with `select:` then the author can specify components to make up the generated text using key words defined by the stylesheet.
- Otherwise the attribute value is taken to be a named cross reference style that is defined in the stylesheet's collection of gentext templates.

Each of these methods is described in more detail in the stylesheet documentation.

New DocBook capabilities

Olinks in the DocBook XSL stylesheets enable several new capbilities in DocBook.

Modular XML files

The DocBook DTD supports large documents, all the way up to sets of books. If you want to form cross references among the books in a set using ID/IDREFs, then the set has to be in a single document. This is so unwieldy that most content managers break up the set into separate book system entity files, and only form the complete set at processing time.The individual books may be further broken down into chapter system entities.

System entities are not complete XML documents. If they contain a DOCTYPE declaration, then this will usually generate an error message when the system entity is pulled into the master document and processed.

The XInclude feature of XML permits each modular file to be a complete XML document. The master document uses <xinclude> elements, each of which uses an XPath expression to point to the root element of a modular file. This avoids the problem of the DOCTYPE declaration, which is left out of the inclusion.

Although each modular file may be a complete XML document, it still may not validate. If it has an `xref` or `link` element whose `linkend` attributes points to an ID in another file, then the modular document is not valid on its own. Both xref and link use the ID/IDREF mechanism, so the linkend must be in the current document.

If all the cross references to targets outside a modular component are converted to `olink` elements, then the file can be validated. That is because the `targetdoc` and `targetptr` attributes are not ID/IDREF, and so they won't be checked. The resolution of olinks takes place when the document is processed with a target database, not when the file is validated against the DTD.

Using olinks means a modular file can be processed on its own and produce reasonable output. Any links within itself would be resolved using ID/IDREF, and any links outside of itself would be written as olinks and resolved using the target database.

Of course, when a modular file is processed on its own, certain context information would be lost. The third chapter in a book would not know it was the third chapter when processed by itself, so its chapter number appear as "1". Likewise, all chapters would begin on page 1. But context information can be provided by the target database to correct these problems. The target database for the master document has all the information that's needed to establish the context for the current modular file. The stylesheet could be extended to supply the `<div>`'s `number` attribute. That number was generated the last time the target data file was generated for the whole document.

Although modular processing is convenient for development, it is not necessary when rendering the final document. To ensure complete sychronization among the modules, the master document can always be processed whole. This way you have reasonably accurate rendering in modular mode, and completely accurate rendering for final output. And it is the master document's target data file that would be used by other documents for linking.

Asynchronous processing

Asynchronous processing means not having to process all your content at once to get cross references to work. You can use the target data from a previously processed document to form a link. The trick is to ensure that the target data won't change again after you process your document with the link to it.

If you are cross referencing among a collection of documents you are developing, then you have control over the target data. To generate a target database, you can freeze the documents and collect the target data. You can do this with a lock or a time stamp in a revision control system, or by simple agreement among developers. Each data collection becomes the reference point for the next cycle of development. At some point the collection is processed for a final time, the output is published, and the target database is archived for future reference.

An author can build their own output as they need to without having to build everything. Different departments or groups can work on their own documents, as long as they regularly synchronize the target data.

Going beyond parallel develop is developing documents in sequence. Later documents can safely make references to documents that have been published and their target data files frozen. You can even cross reference to documents whose source you have never seen, as long as you have access to the final target data file.

Links to open source doc

Olinks managed with this scheme permit you to cross reference to open source documents that are authored in DocBook XML. Often open source documents are the most authoritative on a subject, or at least on the software they document. Most are published in isolation, even if they are dependent on other software, since they have no means of connecting to other documents. Many could at least make use of active links to reference pages for the operating system they run on.

Open source software can also be assembled like building blocks into larger applications that integrate the components into a system. The documentation for the application needs to cross reference to the component docs. Without such links, the application doc needs to duplicate the information, which introduces more development and maintenance work for the authors.

If the DocBook source files are available, and you are rendering those docs, then you can extract the target data yourself to add to your target database. Since open source docs are generally versioned, you can easily tell which version of content you are linking to. Since you are creating the rendering, you know the rendering version as well.

The distributor of the doc will often ship rendered versions along with or instead of the XML source. If the distributor builds a target data file for each rendering to include in the distribution, then that can be added to your target database. Even if the XML source is not included in the distribution, links can be made to the rendered docs.

If the target data is produced as a side effect of rendering the document (controlled by a command line parameter), then the data can be generated regularly. When the final rendering is done for release, the data file can be simply included.

Faster XML web service

With the delays in getting general support for styling XML directly in web browsers, many content providers want to convert XML to HTML on the fly in the web server. But if your XML is a DocBook book, then the user will likely experience significant delays while the whole book is processed. There will be even greater delays while the entire book is delivered as HTML. You can chunk the output to reduce delivery time, but you still must process the whole document each time to resolve ID/IDREF cross references.

These delays can be reduced by breaking up the large book files into smaller modular files. and using olink with a document target database to resolve cross references within the document. Also, the target database can provide the context for chapter numbering and such for modular files. It is much faster to parse the target data file than it is to parse the entire book to establish the context.

So an XML repository that consists of small modular XML files linked together with olink and a target database could be rendered to HTML on demand. Whenever a document module in the repository is updated, it also has its target data updated in the database. When another document that targets the changed document is requested and processed, it uses the updated data to render the links.

Language fallback

When managing a collection of documentation, it is often the case that only certain documents are translated into other languages. If your documents are cross referencing among themselves, then some attention must be given to this issue. You don't want to form a cross reference to a translated document that does not exist.

This problem is solved by properly assembling your target database for your translated documents. The target database supports duplicate values of the `targetdoc` attribute when there are also `lang` attributes

to distinguish among them. That permits you to create a database with references to all language versions of a given document.

When you process your olinks, the stylesheet first tries to find a link in the same language as the document containing the olink (as specified by its `lang` attribute). If the link is not found in that language, then it tried other languages as specified in the stylesheet parameter named `olink.lang.fallback.sequence`. That is a space separated list of language codes that specifies the sequence of languages to try if the first fails.

This makes it easier to manage projects where not all documents are translated at once. As a translation becomes available, it can be added to the database. Then the next time a document with olinks is processed, that data is available for inclusion. But the fallback language is there until the translation is available.

Conclusions

This document has described the implementation of olink processing in the DocBook XSL stylesheets to support cross referencing between documents. The keys points of this scheme are:

- It simplies the syntax to form an olink.
- It moves most of the olink information and processing outside the XML documents and into the XSL stylesheet.
- It merges target information from several documents into a target database document.
- It is designed to render links to rendered documents, thus avoiding the complexity and general lack of support for XLink processing.
- It has been implemented today in the DocBook XSL stylesheets.

For further information, see the chapter on olinking in *DocBook XSL: the Complete Guide [http://www.sagehill.net/docbookxsl/index.html]*