<Submitted to: Open Publish 2001>

Cover Page

Title: **Improving Web linking using XLink**

Length 6782 words

Contact Author: **A/Prof David Lowe**
Affiliation: University of Technology, Sydney
Street Address: No. 1 Broadway, Ultimo
City, State: Sydney, NSW
Postal/Zip Code: 2007
Country: Australia

Telephone: +61 2 9514 2526
Fax: +61 2 9514 2611
Email: david.lowe@uts.edu.au

Additional Author: **Dr. Erik Wilde**
Affiliation: Swiss Federal Institute of Technology, Zürich
Email: net.dret@dret.net

Topic Keywords: XLink
XPointer
linking
transclusion

Improving Web linking using XLink

A/Prof David Lowe
University of Technology, Sydney
david.lowe@uts.edu.au

Dr. Erik Wilde
Swiss Federal Inst. of Technology, Zurich
net.dret@dret.net

ABSTRACT

Although the Web has continuously grown and evolved since its introduction in 1989, the technical foundations have remained relatively unchanged. Of the basic technologies, URLs and HTTP has remained stable for some time now, and only HTML has changed more frequently. However, the introduction of XML has heralded a substantial change in the way in which content can be managed. One of the most significant of these changes is with respect to the greatly enhanced model for linking functionality that is enabled by the emerging XLink and XPointer standards.

These standards have the capacity to fundamentally change the way in which we utilise the Web, especially with respect to the way in which users interact with information. In this paper, we will discuss some of the richer linking functionality that XLink and XPointer enable – particularly with respect to aspects such as content transclusion, multiple source and destination links, generic linking, and the use of linkbases to add links into content over which the author has no control. The discussions will be illustrated with example XLink code fragments, and will emphasise the particular uses to which these linking concepts can be put.

INTRODUCTION

The linking model that underpins the Web is an integral part of HTML and has played a substantial role in supporting the incredible growth of the Web over the last decade. The original model (a simple link from one piece of information to another) was very simplistic – a characteristic that led to much of its success, but which is also now restricting the functionality that can be achieved with the Web. Much more sophisticated linking has been exhibited in a number of other (usually stand-alone) hypertext systems. Many of these systems (and their associated models) are indeed much older than the web. The web's linking model was already outdated (in a technical sense) when it was created. Some of the functionality can be enabled with the Web – but usually only with complex non-standard coding.

The emergence of XML, and in particular the linking model that accompanies it – XLink and XPointer – will enable a much more sophisticated linking model to be natively supported on the Web. In this paper, we will look at some of the emerging possibilities that are enabled by XLink (DeRose et al, 2000) and XPointer (DeRose et al, 2001). We begin by considering typical link functionalities that are not possible with the current Web. We will then move on to look at how they might be supported using XLink and XPointer, and finish up with a typical scenario that highlights some of these uses.

This paper is not intended to be an introduction to XLink and XPointer. Indeed we assume a least a passing knowledge of these standards (though the paper should make sense even without this). Rather, the goal is to illustrate how these standards can support more effective usability of the Web by providing enhanced linking mechanisms.

LINKING FUNCTIONALITY

The linking model that has underpinned the traditional Web model is very simplistic. Essentially, the standard <A HREF="…"> link is a simple static, directional, single-source, single-destination link that is embedded into the source document. It is *static* because the link never changes (at least not once it has been coded into an HTML page). It is *directional* because the link has an explicit direction of association (and hence, usually, an explicit direction of navigation). It is *single-source* because the link only has one point from which it can be triggered. It is *single-destination* because the link only has one resource to which the client traverses when the user activates the link. The link is embedded into the source document (indeed, it is embedded within the source anchor) because the description of the connection between the source and destination anchors exists within the source anchor.

Even this simple model highlights a number of different linking concepts – which in turn indicate aspects that, if changed, can lead to more complex and sophisticated linking. To start with, we have *resources* – an addressable unit of information or service. In other words, basically anything that we are able to request, including Web pages, XML documents, images, audio clips, program outputs, etc. A resource (in the URI sense of the word) is not necessarily a computer-related thing but could, at least conceptually, be anything you would like to deal with, such as paper documents or animals or whatever. Indeed, by utilising standards for addressing fragments of these things, we can also treat fragments as resources. It is worth noting that Web standards - in particular RFC-2396 (Berners Lee et al, 1998) – distinguishes between addressing a whole resource (referred to as an *identifier*, and typically implemented using Universal Resource Identifiers) and addressing a particular sub-resource (implemented using resource-type-specific fragment identifiers).

A resource (or sub-resource) can be used in various ways, but in the context of this discussion, the most common use will be related to linking. In order to do this, we need to define the relevant regions of the resources (or resources themselves) that are participating in a link. Within the context of a link, these regions are referred to as *anchors* – or in XML linking, they are referred to as *locators*.

We also need to distinguish between *links* and *arcs*, which in turn requires an understanding of a third concept - *traversal*. In an HTML A link, we have a single source anchor, a single destination anchor, and an implied connection between them. When we view an HTML document and activate the anchor (in most user interfaces by simply clicking on it), then the link is traversed to the link destination. Other forms of linking in HTML work differently. For example, the link between a document and an image (specified using the IMG element) is traversed automatically when the document is loaded, and the resultant image is embedded into the source. The link between a document and a style sheet (specified using a LINK element) is traversed automatically, but does not result in any content being embedded. In each case, the various link characteristics (such as how the traversal of the link is initiated, the behaviour upon traversal, and the specification of the link semantics) are typically implicit for that link type.

The situation is somewhat different - and much more flexible – with linking in XML. XLink allows the definition of links, but a link does not imply traversal! Rather, in XLink a link is simply an association between a number of resources (specified using locators). There is no link “source”, nor is there a link “destination”. This is because XLink has separated the concept of associating resources (using a link) from the concept of traversal amongst these resources. This traversal is specified using *arcs*. A given link may contain a number of different arcs. Similarly, in many cases we may not need to provide traversal information – especially where the association is being defined for a reason other than to support navigational hyperlinking (for example, to define a collection of resources to be analyzed by computer, rather than to be viewed by a human user). We’ll return to this point again a little later in the paper.

So, using the above concepts, we can identify a number of more sophisticated forms of linking. For example, consider the situation where we have a link that has multiple participating resources, with an arc that starts from more than one resource, and ends on more than one resource. In other words, the arc traversal can be activated from multiple different locations, and when it is activated, it results in the presentation of multiple new resources rather than just a single resource. The result is a multi-source, multi-destination link – something that is not possible to easily implement using the simple linking model within HTML.

We can also have dynamic links, where the destinations resource(s) are only resolved when traversal occurs. Specifically, a dynamic link will have structure or behaviour that changes over time, or with different users or circumstances. These changes can be with the link structure (such as changes in the link destinations) or with the link semantics (such as whether the activation results in the new resource replacing the existing content, or being embedded into the existing content). The most common example of dynamic links are where the link destination is a service rather than a document – such as a CGI script. These types of links can be important for supporting adaptive systems. For example, we may wish to change the destination of a link depending upon the pages a user has previously visited. Although HTML has no inherent support for dynamic links, it is possible to create them using server-side technologies such as CGI scripts or servlets, possibly in conjunction with other technologies such as HTTP cookies or URL rewriting.

A little more interestingly, generic links are links where participating resources are defined not by a specific location, but rather by a particular pattern that can be matched to any relevant content. For example, we could have a link from any occurrence of the word “XML” within a document, to a definition of the term XML. This has several implications that would be useful to look at briefly. The first is that because the source anchor is defined for a particular pattern, rather than a particular location, it would be more accurate to refer to it as a generic anchor (or generic sub-resource) rather than a generic link. Again, implementing generic links with the conventional HTML linking model is very difficult (though possible, given some complex back-end processing).

The above example illustrate some of the limitation of the existing HTML linking model – and hence some of the potential areas that can be improved with XML linking. We will now look in more detail at the XML linking model and how it supports some of these concepts.

XLINK AND XPOINTER

The linking model for XML revolves around the complementary technologies of XLink and XPointer. XPointers provide the mechanism for identifying resource fragments, and XLink provides the mechanism for collecting these together into links.

XPointer

Let's begin by looking at XPointer and its capabilities. XPointer provides a general way to select fragments of an XML document, essentially by writing a set of expressions. An expression is evaluated with respect to the current context (which includes aspects such as bindings between variables and values, and a context element within the given XML document), and usually results in a set of locations (not surprisingly, referred to as a *location set*). An expression can be used to select children, siblings, nodes with given attributes, arbitrary text strings, etc. For example, the XPointer

```
xpointer(//child::body[position()=1]/child::p)
```

selects all p children elements of the first body element in the document.

As another example, the XPointer expression

```
xpointer(/descendant::*[attribute::name='book'])
```

selects all elements (ie, all descendants of the document root) that have an attribute called name with a value of book. In effect, the selection mechanisms can be concatenated to progressively identify a specific subset of nodes.

It is also worth briefly pointing out that XPointer is actually an application of the XPath standard. XPath was developed specifically to be a foundation for other standards such as XPointer. Another example application of XPath is as part of XSL Transformations (XSLT). In XSLT, XPath expressions allow the selection of specific nodes that can then be transformed into an alternative form (often for presentation).

Since XPath is intended to be relatively generic (to suit multiple applications), there are certain sections of documents that cannot be specified using XPath expressions. For example, both of the XPointer fragments shown above are constructed from valid XPath expressions. XPath cannot, however, select an arbitrary string that crosses several nodes. It is in areas such as this that XPointer has extended XPath. For example, the following expression defines the set of all occurrences of the string "links and anchors" within all para elements in the http://a.b/c/d.xml resource (this could not be achieved using just XPath expressions):

```
http://a.b/c/d.xml#xpointer(string-range(//para,'links and anchors'))
```

As one further example, the following URI defines a range that extends from the beginning of the element with an ID of sect-2.3, to the end of the element with an ID of sect-3.4.

```
http://a.b/c/d.xml#xpointer(id('sect-2.3')/range-to(id('sect-3.4')))
```

Note that in this case, this may include only parts of nodes (for example, part of a Chapter-2 element, and part of a Chapter-3 element) – again, something not possible to identify with XPath.

Links and Arcs

It is also worth noting that, as shown in several of the above examples, XPointers are used with URIs to identify a particular resource, and then a fragment of that resource (which must be an XML document). In the context of linking, this means that XPointers can be used to specify anchors – or arbitrary sections of resources that will participate in a link. In XML however, they are referred to as *locator* elements, rather than anchors. A locator element is however more than just an XPointer – we can also specify the role that the resource will play and provide a title for the resource. For example, consider the following XLink fragment:

```
<siblings xlink:type="extended">
  <child xlink:type="locator"
         xlink:href="people.xml#xpointer(id('anna'))"
         xlink:title="Anna"/>
  <child xlink:type="locator"
         xlink:href="people.xml#xpointer(id('bill'))"
         xlink:title="Bill"/>
  <child xlink:type="locator"
         xlink:href="people.xml#xpointer(id('carl'))"
         xlink:title="Carl"/>
</siblings>
```

In this example, we define three locator elements, each of which uses an XPointer as part of the locator required to specify the remote resource that is participating in the link. We also give each locator element a title. We do not, however, specify any traversal between these elements. There is no link "source", nor is there a link "destination". This is because XLink has separated the concept of associating resources from the concept of traversal amongst these resources.

Where we do want to specify traversal information, this is done separately from the specification of the association through the use of an arc. For example:

```
<person xlink:type="extended">
  <name xlink:type="locator"
        xlink:href="staff.xml#xpointer(string-range(/,'David Lowe'))"
        xlink:label="src"/>
  <details
        xlink:type="locator"
        xlink:href="David.xml"
        xlink:label="dest"/>

  <go   xlink:type="arc"
        xlink:from="src"
        xlink:to="dest"/>
</person>
```

In this case we specify an association between two resources (defined by locators): the first resource is all occurrences of a given string ('David Lowe') within one XML document, and the second is another XML document (about David Lowe). We then specify the traversal semantics using an arc from the first resource to the second resource. What this effectively means is that we now have separate mechanisms for specifying an association between resources (a link), and for specifying how we might traverse between these resources (an arc within that link).

Indeed, we can have a single link involving a number of resources, with multiple different traversal rules. Consider the following example, where we have two arc specifications. Also, note that the second arc specification actually creates three arcs since there are multiple destinations specified by the *to* label. We end up with one arc from a.xml (to b.xml), and three arcs from b.xml (to c.xml, d.xml, and e.xml).

```
<extendedlink xlink:type="extended">
  <loc xlink:type="locator" xlink:href="a.xml" xlink:label="x"/>
  <loc xlink:type="locator" xlink:href="b.xml" xlink:label="y"/>
  <loc xlink:type="locator" xlink:href="c.xml" xlink:label="z"/>
  <loc xlink:type="locator" xlink:href="d.xml" xlink:label="z"/>
  <loc xlink:type="locator" xlink:href="e.xml" xlink:label="z"/>

  <go xlink:type="arc" xlink:from="x" xlink:to="y"/>
  <go xlink:type="arc" xlink:from="y" xlink:to="z"/>
</extendedlink>
```

It is also worth noting that XLink can be used to specify the existence of a link without specifying rules for how that link will be used. XLink does support some attributes for defining behaviours – such as how and when an arc should be traversed – but these are optional and when present their interpretation is left to the applications using the XML documents. For example, where multiple arcs emanate from one resource, and that resource is “activated”, the standard does not say how the application should respond. Possible alternatives include traversing all arcs, giving a user the choice of which arc to traverse, or using some internal logic to make the choice. This is illustrated by the above example, where if the arc from b.xml is activated, it is unclear what should be the result.

One mechanism that is supported by XLink is the inclusion of arc roles. For example, consider the following:

```
<extendedlink xlink:type="extended">
  <loc xlink:type="locator"
       xlink:href="a.xml"
       xlink:label="x"/>
  <loc xlink:type="locator"
       xlink:href="b.xml"
       xlink:label="y"/>

  <go xlink:type="arc"
      xlink:from="x"
      xlink:to="y"
      xlink:arcrole="http://q.r/s.dat"/>
</extendedlink>
```

In this case the arc has an arcrole attribute that provides a unique role identifier. This identifier may allow the application to obtain information that assists in determining the appropriate behaviour when traversing the arc. This however is beyond the XLink specification and is application dependent.

Linkbases

Another interesting issue is that of adding links from read only material. This is a rather unusual concept for people who are only familiar with the Web, where all links must be embedded into the source content. This however is very restrictive. For example, we might want to be able to annotate material that doesn’t belong to us with our own links, or the material may be stored on read only media, or we may want to use different sets of links at

different times (depending upon what we are trying to do). In each case, we don't want to, or cannot, add links directly into the underlying content. Instead we would like to be able to specify links separately and somehow have them used. Using XLink, this is relatively straightforward. Consider the following relatively simple example:

```
<?xml version="1.0"?>
<!DOCTYPE Dictionary SYSTEM "Dictionary.dtd">
<Dictionary>
  <Entry word="Anchor">
    <Pronunciation>...</Pronunciation>
    <Definition>An identified region of a node that can be
       explicitly addressed and identified within the presentation
       of a node.
    </Definition>
  </Entry>
  <Entry word="Link">
    <Pronunciation>...</Pronunciation>
    <Definition>A connection between multiple anchors (and nodes,
       where there is an implied anchor that encompasses the entire
       node) that represents an association between the concepts
       captured by the anchors.
    </Definition>
  </Entry>

  <!-- Further entries go here -->
</Dictionary>
```

In this example, the document contains a series of words and definitions – but no links. We can then create a separate file (often called a linkbase) that contains links from words to the definitions of these words.

```
<?xml version="1.0"?>
<!DOCTYPE XRefs SYSTEM "XRefs.dtd">
<XRefs>
  <Xref xlink:type="extended">
    <word xlink:type="locator"
          xlink:href="#xpointer(string-range(//Definition,'anchor'))"
          xlink:label="src"/>
    <defn xlink:type="locator"
          xlink:href="Dict.xml#xpointer(//Entry[@word='Anchor'])"
          xlink:label="dest"/>

    <go   xlink:type="arc"
          xlink:from="src"
          xlink:to="dest"/>
  </Xref>
  <!-- Further cross references go here -->
</XRefs>
```

This linkbase file contains a series of XLinks, which link any occurrence of specific words in the definitions to the definition of those words. For example, the word anchor appearing in the definition of the word link would be the starting point for a link to the definition of anchor. In this case, the links are termed third-party links. This is because they are not embedded into any of the anchors that are participating in the link.

The problem then arises as to how we ensure that these link definitions are actually utilised. The simplest way is, where possible, to modify the source information, so that it includes a reference to the linkbase. This is supported by XLink using a special form of

extended link (ie, it contains an arc from the content to the linkbase, with an arcrole attribute with a special value to indicate that the link destination is a linkbase). When the XML document is viewed or processed, the link to the linkbase will be traversed and the linkbase will be loaded. Even more complex, linkbases can include links to other linkbases, creating hierarchies of linkbases! This solution is fine where we have access to the source information, so that we can add a link to our linkbase, but one of the benefits of XML linking is that we can define third-party links for content that we do not have access to edit. So what do we do in this situation? One solution would be to simply allow the user to specify directly within the browser (or whatever other tool we are using to view or process the documents) the linkbases that we wish to use. This is analogous to the functionality supported in some Web browsers of being able to specify a particular style sheet to use for presentation of Web pages.

Multi-ended links

Another aspect supported by XLink but not supported by HTML linking, is multi-ended links. (These have been shown in a number of the above examples, but not really explained yet). This type of link has a number of important applications – essentially wherever we have more than two resources that participate in a relationship. There are, however, several ways in which we can create links of this type. Firstly, consider the following example (adapted from the XLink standard):

```
<family xlink:type="extended">
<loc xlink:type="locator" xlink:label="parent" xlink:href="Ann.xml"/>
<loc xlink:type="locator" xlink:label="parent" xlink:href="Bob.xml"/>
<loc xlink:type="locator" xlink:label="child" xlink:href="Gina.xml"/>
<loc xlink:type="locator" xlink:label="child" xlink:href="Hank.xml"/>
<loc xlink:type="locator" xlink:label="child" xlink:href="Irma.xml"/>

<go xlink:type="arc" xlink:from="parent" xlink:to="child"/>
</family>
```

In this example, we have five participating resources. We also have a single arc specification which results in six arcs (Ann–Gina, Ann–Hank, Ann–Irma, Bob–Gina, Bob–Hank, Bob–Irma). In effect, we have three arcs from Ann.xml. If we initiate traversal of the arcs that originate from this resource, then XLink does not specify what should happen – though a typical behaviour might be to provide the user with a list of the possible destinations and allow them to select the appropriate arc to traverse. It is also worth noting that XLink not specify how to trigger a traversal – this is largely left to XLink applications (except for the standard behavior attributes).

XLink also supports a second form of multi-ended link – though it is a little more subtle than the above example. Consider the following:

```
<family xlink:type="extended">
  <loc xlink:type="locator" xlink:label="parents"
       xlink:href="Family.xml#xpointer(//Person[@type='parent'])"/>
  <loc xlink:type="locator" xlink:label="children"
       xlink:href="Family.xml#xpointer(//Person[@type='child'])"/>

  <go xlink:type="arc" xlink:from="parents" xlink:to="children"/>
</family>
```

In this example we have a single arc specification which results in just a single arc – so how can this be a multi-ended link? The key is in the XPointer. Essentially, the XPointer

selects a location set, which in the case of the parents locator, can potentially be a set of multiple non-contiguous elements. The same applies to the children locator. In effect, we have a single arc from one sub-resource to another sub-resource – but both sub-resources can potentially be location sets with more than one member.

Again, XLink does not specify how this situation, once specified in XLink, should be interpreted in terms of navigational behaviour. In particular, the standard states:

> "Some possibilities for application behaviour with non-contiguous ending resources might include highlighting of each location, producing a dialog box that allows the reader to choose among the locations as if there were separate arcs leading to each one, concatenating the content of all the locations for presentation, and so on. Application behaviour with non-contiguous starting resources might include concatenation and rendering as a single unit, or creating one arc emanating from each contiguous portion."

Before we finish this section, it is also worth pointing out that XLink supports annotating arcs with additional semantics – specifically titles and roles. We can have two arcs between the same two resources, but with different purposes as indicated by their roles.

Generic Links

Another concept worth looking at is generic linking. A generic link is effectively a link that has a participating resource that is specified not by defining the actual resource, but by specifying a set of conditions that must be met for a resource to be included. This can be readily achieved using the functionality of XPointer. In particular, XPointer allows the selection of resource fragments based on criteria such as

- elements of a given type,
- elements that have a specific attribute with a given value,
- text that matches a given string, or
- complex combinations of the above.

In each case, the resulting location set can contain multiple locations. The result, when the XPointers are used in XLink, is a specification of a generic fragment identifier. This can be the source or the destination of possible arcs. If used as the source, then we have the conventional generic link as described previously. If used as the destination of an arc, then we potentially have a link that traverses to an aggregation of all references to a particular concept or element. It is worth pointing out that this allows the specific of generic sub-resources (i.e. genericity within a particular resource) but not really generic resources (since the URI cannot contain a generic component).

This type of XLink linking could be expected to be most common for producing universal cross-referencing, such as links from words to their definitions in dictionaries or glossaries.

Typed Links

Finally, one last aspect that is not supported by HTML linking, but is supported by XLink, is the concept of typed links. Essentially, a typed link belongs to a particular set of links with common characteristics. The fact that it belongs to that set is typically indicated by the link having a particular attribute with a given value.

The ability to type links can make it much easier to navigate through a web of interlinked resources. For example, we can "switch" certain links on or off, or request certain types of

links to be highlighted. For example, when initially learning a particular concept we may wish all glossary links to be visible, so that we can readily obtain definitions of unfamiliar terms. As we begin to understand the topic, we may wish to switch off the glossary links to reduce clutter. Apart from the visibility of links (or rather, resources participating in the link), we might wish to change the traversal behaviour of certain link types: changing whether or not confirmation is required, where the destination is displayed, etc.

It is also worth noting that since XLink separates the concepts of association (ie, links) from traversal (ie, arcs) we can also type these two elements independently. In other words, we can have various link types, but we can also support various arc types. So a given link might contain multiple arcs (possibly even between the same resources) with different types.

So how do we actually do this typing in XLink? Although not explicitly designed for link typing, the standard provides several semantic attributes that effectively provide this support. In particular, both the title attribute and the role and arcrole attributes can be used to support link typing. The title attribute is typically used to describe the meaning of a link or resource in a human-readable form. The intended use of the title is not constrained by the standard, but is likely to used for purposes such as making titles available to visually impaired users, generating tables of links, or to provide help text in the form of tool tips before an arc is traversed. Although we could type links by defining specific titles to use for specific link types, this would be a somewhat cumbersome way to support typing, and would be likely to interfere with other uses of the title attribute.

The role and arcrole attributes are, however, more appropriate for link typing. The value of these attributes (according to the standard) must be a URI reference that identifies some resource that describes the intended property. We could therefore define a URI that is to be used by all links or arcs that belong to a particular type. Indeed, we could support links or arcs belonging to multiple types by having the role and arcrole URIs point to a resource that contains a list of the types for that link or arc. Consider the following example:

Family.xml:

```
  ...
  <loc xlink:type="locator" xlink:label="p1" xlink:href="Ann.xml"/>
  <loc xlink:type="locator" xlink:label="p2" xlink:href="Bob.xml"/>
  <loc xlink:type="locator" xlink:label="c1" xlink:href="Gina.xml"/>
  <loc xlink:type="locator" xlink:label="c2" xlink:href="Hank.xml"/>
  <loc xlink:type="locator" xlink:label="c3" xlink:href="Irma.xml"/>

  <go xlink:type="arc" xlink:from="p1" xlink:to="c1"
      arcrole="http://transclude.com/demo/4682.arc"/>
  <go xlink:type="arc" xlink:from="p2" xlink:to="c1"
      arcrole="http://transclude.com/demo/8634.arc"/>
```

4682.arc:

```
  arc-type: mother
  arc-type: guardian
```

8634.arc:

```
  arc-type: father
  arc-type: guardian
```

In this case, we have constructed a (somewhat arbitrary and trivial) format for the .arc files simply to illustrate the point. For astute readers, however, the above (simplistic) example might point towards a potential problem. The effective use of link typing requires that a

common vocabulary is used for describing the link types. The XLink standard does not make any suggestions in this direction (indeed, this particular use of role and arcrole attributes is not discussed). We could potentially adapt a standard such as the Dublin Core (Weibel et al, 1998), which provides a standard set of metadata, to specifying attributes of the link or arc. We would constrain the attributes that we could use, but not their values. More generically, we could utilize RDF, but again, we run into similar problems. RDF is more general than the Dublin Core – not defining a fixed set of attributes, but rather a mechanism for defining a meta-data schema – but we would still need to develop a suitable schema. At present, these are issues that are yet to be resolved.

LINKING SCENARIO

To demonstrate XLink a little further, we will look at a typical scenario that would benefit immensely from a more sophisticated linking model – showing how XLink and XPointer might be used.

Jack is undertaking an online correspondence course about XML and linking. The course includes support for richly linked course content, discussion forums, interactive groupwork, etc. Jack is completing some work for the course and after logging in to the main site, goes to the course material for the relevant session on linking concepts. The material provides an initial discussion of some basic concepts and then directs Jack to study a set of readings that have been published elsewhere on the Web (and predate the course material, hence the authors of these readings are oblivious to the existence of the course that Jack is taking).

Jack follows the link to the first article and commences reading. Whilst reading the material he comes across the term "transclusion". He clicks on this word and a window pops up providing Jack with a definition of transclusion provided by his instructor.

Adding links to read-only material. In this case, the course instructor has added links for definitions into material over which she has no control – material that (as far as the instructor is concerned) is read-only. With conventional HTML pages, links must be embedded into the source material, which in this case cannot be modified. As such, the only ways of achieving this using standard Web servers and HTML pages would be to either save the pages locally and modify them to include the new links (which is inefficient and has copyright problems), or to extend the server functionality to allow it to obtain other pages, modify them dynamically to add new links, and then deliver the modified pages (effectively a link-adding proxy – again, rather complex to manage).

This difficulty is a consequence of having to embed links into the source material. A much simpler approach would be to be able to define links independently of the source material. A user could specify (or have specified for them) a set of pages to view, and the list of links to use with this content (potentially stored completely independently from the content). The result would be to allow authors to add their own links into content over which the author has no control. Indeed different lists of links could be created for different users or different situations. As will be shown later, XLink explicitly supports this type of functionality through the use of third-party links and external linkbases.

Jack reads the definition and then closes the pop-up window. He continues reading only to come across the phrase "Ted Nelson's definition of transclusion includes the concept of contexts". Again, Jack selects "transclusion" but this time a menu pops-up providing Jack

with the choice of four link destinations. The first choice is the definition which he has already seen.

Generic Links. This part of the scenario described a link that had the same link anchor text as the previous example (ie, “transclusion”). We can create this situation by simply adding a new link anchor (and associated link) for each occurrence of the relevant phrase, but this could become extremely cumbersome and difficult to maintain if the word occurred often.

The use of generic links solves this problem. By defining a link that has as its source anchor any occurrence of the relevant text, we effectively create a link that is much easier to maintain. Again, this is relatively easy to implement in XML. For example, the following XPointer refers to any occurrence of the text “transclusion” within element content in a document (though it won’t match attribute values and any XML markup such as element or attribute names):

```
xpointer(string-range(//*,'transclusion'))
```

This XPointer can then be used to create a link from all occurrences of this text to the relevant definition of this text in a file containing a list of definitions. For example, the following third-party extended link provides a link from all occurrences of the words “transclude”, “transclusion”, and “transcluding” within the link.xml file to an appropriate definition in the defs.xml file (note that this link can stored in a third file unrelated to either):

```
<extendedlink xlink:type="extended">
<loc xlink:type="locator"
   xlink:href="links.xml#xpointer(string-range(//*,'transclude'))"
   xlink:role="phrase"/>
<loc xlink:type="locator"
   xlink:href="links.xml#xpointer(string-range(//*,'transclusion'))"
   xlink:role="phrase"/>
<loc xlink:type="locator"
   xlink:href="links.xml#xpointer(string-range(//*,'transcluding'))"
   xlink:role="phrase"/>
<loc xlink:href="defs.xml#xpointer(//defn[phrase='transclude'])"
   xlink:role="defn"/>

<go xlink:type="arc"
   xlink:from="phrase"
   xlink:to="defn"
   xlink:show="new"
   xlink:actuate="onRequest"/>
</extendedlink>
```

The second destination is to Ted Nelson’s definition (in its original source) and the third and fourth destinations relate to two different discussions of the relationship between transclusion and context.

Overlapping anchors. The example described is more complex than just using generic anchors. In this case, Jack is given a choice between four possible destinations. These destinations could be generated from several sources. The first destination given is generated from the fact that the selected word (“transclusion”) has a generic link to a definition of the phrase (as described above).

The second destination (to Ted Nelson's original definition and discussion of transclusion) might have been generated from an outbound link embedded into the content itself. For example, the source content for the page may have looked something like:

```
<p id="p23">...so let us consider the issue of context. <simplelink
xlink:href="Nelson.htm" xlink:show="new"> Ted Nelson's definition of
transclusion</simplelink> includes the concept of contexts. What this
means is that...
```

In this example, the phrase *"Ted Nelson's definition of transclusion"* is the anchor for a simple link. But the word "transclusion" within this phrase is also the anchor for a generic link, so when it is selected, the user is given the choice as to which destination they wish to follow.

The situation can become more complex still if we add another third-party link with an anchor that overlaps. For example, we could define the following link (abbreviated for clarity):

```
<extendedlink xlink:type="extended">
  <loc xlink:href="doc0.xml#xpointer(string-range(id('p23'),
            'transclusion includes the concept of contexts'))"
       xlink:role="phrase"/>
  <loc xlink:href="doc1.xml"
       xlink:role="discussion"/>
  <loc xlink:href="doc2.xml"
       xlink:role="discussion"/>

<go xlink:from="phrase"
    xlink:to="discussion"
    xlink:show="new"
    xlink:actuate="onRequest"/>
</extendedlink>
```

This creates a link from the phrase *"transclusion includes the concept of contexts"* within the specified document to two possible destinations. In other words, we have overlapping anchors. We also have a link that has two possible destinations (which in this case the user can choose from). In summary, we have the following:

- A generic (i.e. external) link that creates an anchor from "Ted Nelson's definition of transclusion includes the concept of contexts" for a link to the instructor's definition of transclusion.

- An embedded link that creates an anchor from "Ted Nelson's definition of transclusion includes the concept of contexts" for a link to Ted Nelson's original definition of transclusion.

- An external link that create an anchor from "Ted Nelson's definition of transclusion includes the concept of contexts" for a link to two different discussions of transclusion and context.

When the word "transclusion" is selected, Jack is given the choice of which of the four possible destinations he wishes to view.

Jack finishes reading the article and returns to the main page for the current week's course material. He continues to read several additional articles, including a very recent essay. A

few days later he returns to the course material and navigates to a discussion that compares the articles. The discussion includes fragments from most of the articles.

Transclusion – supporting composition: This fragment of the scenario is a true example of transclusion. The document that is being viewed by Jack could be constructed manually, but it makes more sense to build it directly from the original sources. For example, consider the following XML fragment:

```
...
Another definition of links has been provided by Joe Bloggs. Joe has
stated that:
<simplelink xlink:href="bloggs.xml#xpointer(id('quote32')"
            xlink:show="embed" xlink:actuate="onLoad"/>
...
```

In this case, the content from the remote resource is embedded directly into the document. The show="embed" tag means that the content is viewed directly in the source document rather than being viewed independently. The actuate="onLoad" means that the embedding should occur immediately on loading the source document.

Jack notices that the discussion compares several definitions of the term "link", but the definition from the recent essay seems different from what he remembers. He selects the definition and from the main menu chooses to see the original source. A second window pops-up showing the definition in its original context of the essay, and Jack notices that the definition has indeed been changed in the original material.

Transclusion – supporting access to source: In the "embedding" example just described, the reader need not know that the content has been embedded from a different source. However, in the true spirit of transclusion, it is useful for the reader to be aware of this, so that they can view the material in its original context if desired. In this case, Jack has seen a quote and wants to see its original source. He selects the quote and then selects the appropriate menu option (which is, of course, dependent upon the particular implementation).

The browser could then retrieve the Bloggs.xml document and highlight the section indicated by the XPointer xpointer(id("quote32")). In effect, Jack has been able to see the transcluded content in its original context.

Jack then returns to the original discussion of links, and chooses to see an animation showing how a Web server might support generic links. Whilst watching the animation, he sees a Web server interacting with a linkbase. He is unsure what a linkbase is, and next to the animation is a list of components shown within the animation (server, linkbase, webpages, network, etc.). Jack clicks on "linkbase" and the list of words remains, but the single word "linkbase" is replaced by a short description of the linkbase.

Link-semantics – embedding content: This part of the scenario illustrates an alternative use of embedding content. In this case we have a list of words that forms the basis for anchors. When a word is selected, the relevant arc is traversed and rather than causing a new document to be viewed (either in a new window or replacing the existing document), the content is embedded into the existing document. For example, consider the following fragment:

```
<components>
  <comp>
```

```
      <simplelink
           xlink:href="desc.xml#xpointer(//desc[name='server'])"
           xlink:show="embed" xlink:actuate="onRequest">Server
      </simplelink>
    </comp>

    <comp>
      <simplelink
           xlink:href="desc.xml#xpointer(//desc[name='linkbase'])"
           xlink:show="embed" xlink:actuate="onRequest">LinkBase
      </simplelink>
    </comp>
  ...
  </components>
```

Each word is an anchor for a link. The link however creates embedded content only when it is activated. Thus when Jack selects a component name, the name is replaced by the description of that component. Effectively, this has implemented a partial folding editor.

The animation is also paused.

Link Semantics – controlling context: This final fragment of the scenario illustrates an even more complex situation. In this case, the document contains several media components. When we activate a link in one component of the document (ie, selecting a text name), we wish to cause a change in the behaviour of another component of the document (ie, pausing the playing of an animation). The current version of XLink does not allow behaviour such as this to be explicitly specified, but it does provide an ”extension” mechanism. XLink supports an show="other" attribute value and explicitly states that this could be used to instruct an application to look for other markup determining the presentation. We could then include suitable markup that explicitly defined the behaviour described above – possibly by appropriate use of style sheets (and in particular using the multi-switch and multi-property-set XSL-FO formatting object) or suitable scripting.

Jack reads this, and then clicks on the description. The description is once again replaced by the single word, restoring the list to its original state. The animation recommences and Jack continues watching but now with a better understanding of the interaction being shown.

CONCLUSIONS

In this paper we have attempted to illustrate some of the functionality that will be enabled by the emergence of the XPointer and XLink standards. Although the standards have reached a relatively degree of stability, tools to support these standards are still emerging. It is important to note that much of the semantics of XLink-coded links will be defined outside of the XLink and XPointer standards. As such, although we have detailed a typical scenario, the specific interface behaviours will largely be dependent upon the design of XLink support within browsers and other related tools.

REFERENCES

T. Berners-Lee, R. T. Fielding, and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax. Internet draft standard RFC 2396". August 1998.

S. J. DeRose, E. Maler, and D. Orchard, "XML Linking Language (XLink) Version 1.0. World Wide Web Consortium, Proposed Recommendation PR-xlink-20001220,".December 2000.

S. J. DeRose, E. Maler, and R. Daniel, "XML Pointer Language (XPointer) Version 1.0. World Wide Web Consortium, Working Draft WD-xptr-20010108,". January 2001.

S. L. Weibel, J. A. Kunze, C. Lagoze, and M. Wolf, "Dublin Core Metadata for Resource Discovery. Internet informational RFC 2413", September 1998.

BIOGRAPHIES

Associate Professor David Lowe is the Director of Undergraduate Programs in the Faculty of Engineering, and a Co-Director of the Centre for Object Technology, Applications and Research (COTAR) at the University of Technology, Sydney. He has active research interests in the areas of Web development and technologies, hypermedia, and software engineering. In particular he focuses on Web development processes and web project specification and scoping, and information contextualisation. He has published widely in the area, including several texts (Lowe and Hall, Hypermedia and the Web: An Engineering Approach, Wiley, 1999 and Wilde and Lowe, Transcluding the Web: Linking and XML, Addison-Wesley (currently in preparation)). In the last 7 years he has published over 40 refereed papers and attracted over $900,000 in funding, including a recent grant for research into Web project specifications. He is on numerous Web conference committees and is the information management theme editor for the Journal of Digital Information. He has undertaken numerous consultancies related to software evaluation, Web development (especially project planning and evaluation) and Web technologies. David can be contacted at david.lowe@uts.edu.au.

Dr. Erik Wilde is currently working at the Computer Engineering and Networks Laboratory (TIK), which is part of the Department of Electrical Engineering of the Swiss Federal Institute of Technology (ETH) in Zürich. His interest in general are Web technologies, with a special focus on content-related standards such as XML/XLink/XPointer. When he is not reading new standards (or out in the woods running...), he is giving lectures or courses, trying to make some progress with various book projects, working on a research project dealing with flexibly and effectively applying the topic map metaphor to the Web, or spending time as a consultant for companies seeking help or advice with Web-related projects. Erik can be contacted at net.dret@dret.net.