6 XML Pointer Language

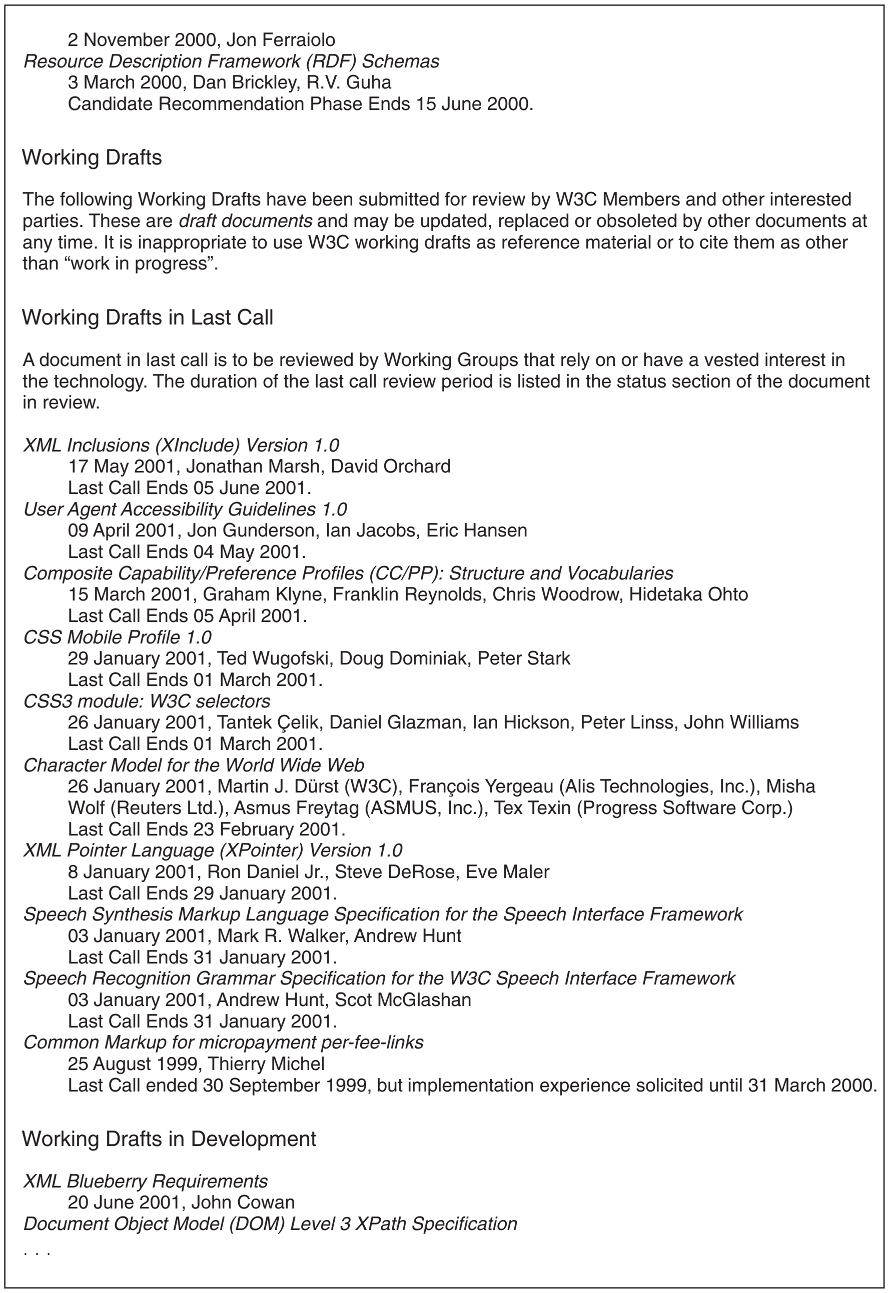
XPath, described in detail in the previous chapter, provides a common foundation for other standards that need to address into XML documents. One such standard, and the most interesting with regard to implementing hypermedia based on XML technologies, is the XML Pointer Language (XPointer) [DeRose+ 01a], which is used for fragment identifiers for XML resources. According to RFC 3023 [Murata+ 01], XML documents are associated with a number of MIME types.[1] For all these different types of XML resources, it is possible to specify a fragment identifier, which is separated from the URI of the resource itself by a crosshatch (`#`) character. As defined in RFC 2396 [Berners-Lee+ 98] (the standard for URI syntax), a fragment identifier is not an actual part of a URI but is often used in conjunction with one in the so-called URI reference.

Thus, XPointer can be used for specifying references that point to parts of an XML document, and not to the whole document. As a simple example, while the URI `http://www.w3.org/TR/` references the technical reports page of the W3C (as shown in Figure 6.1), the URI reference `http://www.w3.org/TR/#xpointer(id('xptr'))` specifically points to the entry for the XPointer standard on that page.[2] This mechanism makes it possible to create links that are much more specific than through the use of URIs only. There are, however, several things to keep in mind, as follows:

- *The resource must be XML.* XPointer is a mechanism for addressing into XML documents, so the resource identified by the URI must be

[1]The Multipurpose Internet Mail Extensions (MIME) [Borenstein & Freed 92] define (in addition to a multitude of other things not relevant in this context) a mechanism for identifying types of resources by means of a media type (e.g., `text`) and a subtype (e.g., `html`).

[2]The XPointer standard may have changed status since the time of this writing and consequently will not appear on the technical reports page as shown in the figure (and as used in the examples in sections 6.4.3 and 6.4.4). However, the general principle does not depend on the particular status of the standard or the W3C's technical reports page.



2 November 2000, Jon Ferraiolo
Resource Description Framework (RDF) Schemas
3 March 2000, Dan Brickley, R.V. Guha
Candidate Recommendation Phase Ends 15 June 2000.

Working Drafts

The following Working Drafts have been submitted for review by W3C Members and other interested parties. These are *draft documents* and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to use W3C working drafts as reference material or to cite them as other than "work in progress".

Working Drafts in Last Call

A document in last call is to be reviewed by Working Groups that rely on or have a vested interest in the technology. The duration of the last call review period is listed in the status section of the document in review.

XML Inclusions (XInclude) Version 1.0
17 May 2001, Jonathan Marsh, David Orchard
Last Call Ends 05 June 2001.
User Agent Accessibility Guidelines 1.0
09 April 2001, Jon Gunderson, Ian Jacobs, Eric Hansen
Last Call Ends 04 May 2001.
Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies
15 March 2001, Graham Klyne, Franklin Reynolds, Chris Woodrow, Hidetaka Ohto
Last Call Ends 05 April 2001.
CSS Mobile Profile 1.0
29 January 2001, Ted Wugofski, Doug Dominiak, Peter Stark
Last Call Ends 01 March 2001.
CSS3 module: W3C selectors
26 January 2001, Tantek Çelik, Daniel Glazman, Ian Hickson, Peter Linss, John Williams
Last Call Ends 01 March 2001.
Character Model for the World Wide Web
26 January 2001, Martin J. Dürst (W3C), François Yergeau (Alis Technologies, Inc.), Misha Wolf (Reuters Ltd.), Asmus Freytag (ASMUS, Inc.), Tex Texin (Progress Software Corp.)
Last Call Ends 23 February 2001.
XML Pointer Language (XPointer) Version 1.0
8 January 2001, Ron Daniel Jr., Steve DeRose, Eve Maler
Last Call Ends 29 January 2001.
Speech Synthesis Markup Language Specification for the Speech Interface Framework
03 January 2001, Mark R. Walker, Andrew Hunt
Last Call Ends 31 January 2001.
Speech Recognition Grammar Specification for the W3C Speech Interface Framework
03 January 2001, Andrew Hunt, Scot McGlashan
Last Call Ends 31 January 2001.
Common Markup for micropayment per-fee-links
25 August 1999, Thierry Michel
Last Call ended 30 September 1999, but implementation experience solicited until 31 March 2000.

Working Drafts in Development

XML Blueberry Requirements
20 June 2001, John Cowan
Document Object Model (DOM) Level 3 XPath Specification
. . .

Figure 6.1 Snapshot of W3C's technical reports page

XML.[3] In the example just given, this is true since W3C makes its pages available in XHTML, the XML variant of HTML. However, the vast majority of documents on the Web are not XML, and consequently XPointer cannot be used to address into them. While it is assumed that XML resources will become more popular in the near future (in particular since XHTML is the successor of HTML), as long as non-XML browsers are still widely used,[4] HTML is likely to remain the most popular language.

As a side note, HTML also supports fragment identifiers, but they are limited to pointing to IDs only (as opposed to the XPath-based addressing capabilities of XPointer). HTML uses its own extremely simple syntax for fragment identifiers, which works by giving the ID as the fragment identifier, so the XML example just given would be equivalent to the HTML version `http://www.w3.org/TR/#xptr`.[5] (In this case, there is a simple correspondence between the XML and the HTML fragment identifier, because both address the fragment using its ID.)

- *The resource must remain available.* Of course, a fragment identifier makes sense only as long as the resource is still available. This brings up the well-known problem of broken links in the Web, and it is independent from specifying fragment identifiers. However, because fragment identifiers are often used with URIs, this issue must be addressed. Resources on the Web often have an astonishingly short life span [Dasen & Wilde 01]; and while some resources disappear (i.e., no longer exist or at least are no longer available via a known URI), others are moved to a new URI without having automatic redirections set up by the Web server operator.
- *The ID must remain the same.* In cases where the fragment identifier uses an ID within the document, it will work correctly only as long as

[3]W3C's technical reports page is being served by the Web server with the MIME type `text/html`, so technically it is not an XML resource. However, on inspection (and validation), it can be concluded that the page indeed is XHTML, so it can be regarded as being an XML resource.

[4]Here it is important to notice that XHTML has been specifically designed to be usable by non-XML browsers (such as older HTML browsers). However, as long as there is only limited market pressure to provide XHTML rather than HTML, most content providers will be likely to continue to use HTML.

[5]If this URI does not work in your browser, try `http://www.w3.org/TR/#xptr` instead, which definitely should work. This, however, is not a mistake of the first variant of the URI or of the server but a sign of a badly implemented browser [Dubost+ 01]. At the time of writing, only Internet Explorer handles this case correctly, while Navigator and Opera get a redirect response from the server and then fail to append the fragment identifier to the URI to which they were redirected.

the ID remains valid within the document (and, in the example just given, continues to identify the element representing the XPointer entry within the document). However, since we do not have control over the W3C's document management and identification policy, we have no guarantee that the ID will always be the same and that it will always identify the element we want to reference. The basic dilemma behind this is that the resource (the W3C's Web page) and the reference to it (our fragment identifier) are handled by different entities, which do not necessarily cooperate (or even know each other; for example, even though we know the W3C's Web pages, the W3C probably does not know that we used their ID as an example in this book).

- *The client has to support XPointers.* Even if all previous requirements are satisfied (i.e., the document is XML, it is still available via the URI, and the XPointer can still be interpreted meaningfully), the application processing the URI with the fragment identifier must implement the XPointer standard. At the time of writing, this is not the case for almost all available software, though we hope this will change in the near future. In comparison to XLink and XPath, XPointer is lagging behind in the standardization process; and as long as there is no stable standard, it cannot be implemented.

  The major browsers in their most current versions (at the time of writing, Internet Explorer 5.5, Navigator 6, and Opera 5) all support XML in the sense that they are able to not only download and interpret XML documents, but also to display them using style sheet mechanisms (CSS and/or XSL). There is, however, no support for XPointer currently. Nevertheless, as soon as XPointer reaches recommendation status, we hope to see XPointer support (as well as XLink support) in the next releases of the major browsers.

These are the requirements that must be met when using XPointer. We believe that in the near future XPointer (along with many other XML-based technologies) will become widely supported and a popular technology. For an illustration of how XPointer may not only become useful for hypermedia applications (which are the focus of this book) but also for other relatively simple cases of usage, consider the following scenario:

> You find an interesting quote on the Web, possibly in an XHTML resource, that you would like to send to a friend. Instead of copying the quote into an e-mail (which would mean taking the quote out of context) or simply sending the resource's URI (which would make it necessary to somehow indicate exactly which part of the resource you mean), you select the quote with the mouse and then choose the "Generate XPointer" option from your browser's menu, which automatically

generates a URI reference that exactly identifies the selected quote. You paste this URI reference into the e-mail and send it to your friend. This way, you have exactly identified the quote that was important to you without taking it out of context. Upon receiving the URI reference, your friend's browser not only requests and displays the resource containing the quote but also automatically highlights the quote identified by the XPointer part of the URI reference.

This example depends on the browser's ability to generate XPointers. Ideally, it would do so in a clever way, because for each subresource there is a multitude of possibilities for creating an XPointer identifying it. We will discuss this important issue in detail later in this chapter (in sections 6.4.3 and 6.4.4), but by now it should be clear that XPointer can provide a lot of value in an XML-based Web.

We now look at the details of XPointer. Section 6.1 discusses the general data model of XPointer, which is a generalization of XPath's data model. After this introductory section, we go into the details of how XPointers may be used as fragment identifiers, described in section 6.2. The next issue is XPointer's extensions to XPath and, in particular, the additional functions that XPointer defines. These functions are discussed in section 6.3.

After this rather formal discussion of XPointer, we then spend some time considering possible usage scenarios and how XPointer may be applied in the best possible way (section 6.4). Finally, even though XPointer is a very new standard, in section 6.5 we briefly describe our view of what XPointer's future may look like.

6.1 GENERAL MODEL

One of the most important aspects of XPointer is that it defines a generalization of the XPath concepts of *nodes, node types,* and *node sets* (as described in section 5.1) to the XPointer concepts of *locations, location types,* and *location sets.*[6] As a reminder, nodes, node types, and node sets in XPath are used to describe concepts that can be identified as nodes in a document's tree representation, as described by the XML Infoset. XPath functionality, such as filtering an axis output by predicate, is generally defined in terms of operations on nodes and node sets (an exception are the string functions, but these are rather limited and always operate on strings within one text node).

XPointer's goal is to define a mechanism for XML fragment identifiers. A very common usage scenario is a user selecting arbitrary document content with a mouse and then wishing to have an XPointer generated that

[6]Despite their similar names, these concepts are completely independent from XPath's location paths and location steps and should not be confused with them.

identifies exactly that content (e.g., to use the XPointer for creating a link pointing to that content). Since this selection can span multiple elements and furthermore may start in the middle of the text of one element and end in the middle of another, it is impossible to identify this content with XPath's constructs of nodes or strings. XPointer's solution to this problem is an extension of XPath's data model, described in section 6.1.1. To make the concepts of XPointer's data model easier to understand, we give some examples in section 6.1.2 of how this model maps to real-world scenarios.

6.1.1 XPointer Data Model

XPointer generalizes the concept of XPath nodes to *locations,* and, in essence, this generalization defines each location to be an XPath node, a *point,* or a *range.*[7] The following definitions are taken from the XPointer specification[8] and show how XPath's definition of a `NodeType` is extended by the concepts point and range:

```
[11] NodeType              ::= 'comment'
                             | 'text'
                             | 'processing-instruction'
                             | 'node'
                             | 'point'
                             | 'range'
```

Based on these definitions, XPointer also defines the *location set* as a generalization of XPath's *node set.* This definition allows XPath node tests to select locations of type point and range from a location set that might include locations of all three types. All locations generated by XPath constructs are nodes, but XPointer constructs can also generate points and ranges. The concepts of points and ranges are defined in the next two sections.

Point

A location of type *point* is defined by a node, called the *container node,* and a non-negative integer, called the *index.* It can represent the location preceding any individual character, or preceding or following any node in the information set constructed from an XML document. Two points are

[7]Note that the XPointer concepts of points and ranges directly correspond to the concepts of *positions* and *ranges* as defined by the Document Object Model (DOM) [Kesselman+ 00].

[8]We list XPointer grammar productions only where they help in understanding the concepts behind them. The numbering of the productions has been taken from the XPointer specification [DeRose+ 01a], which should be consulted for a complete and authoritative definition of the XPath grammar. It can be found at `http://www.w3.org/TR/xptr`.

identical if they have the same container node and index. Each point can be either a *node point* or a *character point,* which are defined as follows:

- *Node point.* If the container node of a point is of a node type that can have child nodes (that is, when the container node is an element node or a root node), then the index is an index into the child nodes, and such a point is called a *node point.* The index of a node point must be greater than or equal to zero and less than or equal to the number of child nodes of the container. An index of zero indicates the point before any child nodes, and a non-zero index n indicates the point immediately after the nth child node.
- *Character point.* When the container node of a point is of a node type that cannot have child nodes (i.e., text nodes, attribute nodes, namespace nodes, comment nodes, and processing instruction nodes), then the index is an index into the characters of the string value of the node. Such a point is called a *character point.* The index of a character point must be greater than or equal to zero and less than or equal to the length of the string value of the node. An index of zero indicates a point immediately before the first character of the string value, and a non-zero index n indicates the point immediately after the nth character of the string value.

Figure 6.2 shows the relationship of container nodes, node points, and character points for an example of an element containing text, then another element (which also contains text), and then some more text.

XPointer's goal is to make XPath's concepts applicable to locations and not only to nodes, and thus the following properties for applying XPath's concepts to points are defined: The `self` and `descendant-or-self` axes of a point contain the point itself. The `parent` axis of a point is a location set containing a single location, the container node. The `ancestor` axis contains the point's container node and its ancestors. The `ancestor-or-self` axis contains the point itself, the point's container node, and its ancestors. The `child`, `descendant`, `preceding-sibling`, `following-sibling`, `preceding`, `following`, `attribute`, and `namespace` axes of points are always empty.

Range

A range is defined by two points, a *startpoint* and an *endpoint.*[9] A range represents all of the XML structure and content between the startpoint and the endpoint. A range whose start- and endpoints are equal is called

[9]The start point has to appear before the end point in document order. More discussion about document order and point locations is at the end of this section.

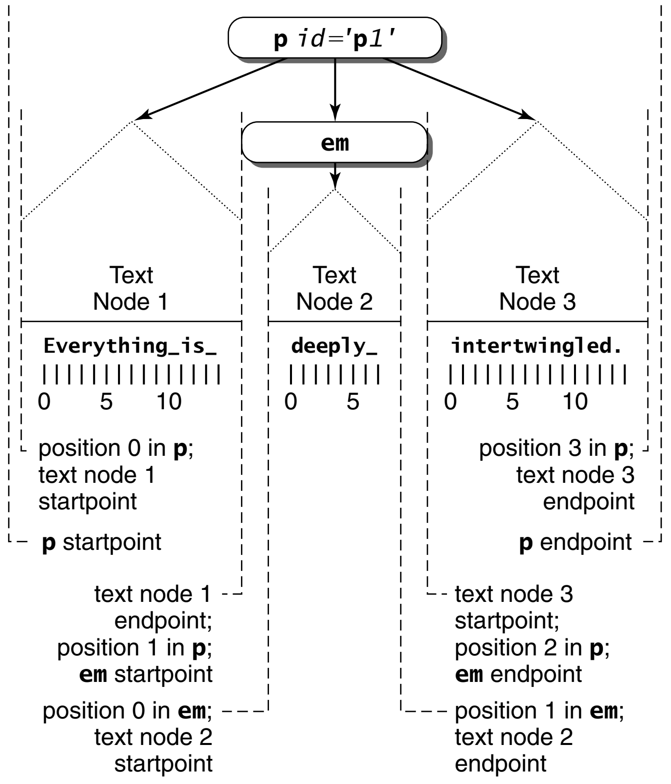

p id='p1'
em
Text Node 1
Text Node 2
Text Node 3
Everything_is_
deeply_
intertwingled.
0 5 10
0 5
0 5 10
position 0 in p; text node 1 startpoint
p startpoint
position 3 in p; text node 3 endpoint
p endpoint
text node 1 endpoint; position 1 in p; em startpoint
position 0 in em; text node 2 startpoint
text node 3 startpoint; position 2 in p; em endpoint
position 1 in em; text node 2 endpoint

Figure 6.2 Container nodes, node points, and character points. Reproduced from the XPointer specification [DeRose+ 01a] by kind permission of Steven DeRose, spring 2002.

a *collapsed range*. If the container node of one point of a range is a node of a type other than element, text, or root, then the container node of the other point of the range must be the same node.[10] The axes of a range are identical to the axes of its startpoint.

As a side note, remember that node sets are only one of the object types defined in XPath (the others being boolean, number, and string), and the other XPath object types also exist in XPointer. However, there is one important exception, and this is the result type of a whole XPointer. While an XPath can evaluate to any object type (consider the rather simple XPath `2+3`, which evaluates to a number), XPointer requires an XPointer

[10]This rule has been defined to make it impossible to create ranges that start within attribute, namespace, comment, or processing instruction nodes and end in a node other than the same node.

to always evaluate to a location set (which is logical, given that there is no way that objects other than location sets could be interpreted as fragment identifiers pointing into documents).

In order to understand some of the functions XPointer provides, it is also essential to introduce the concept of the *covering range*. By definition, a covering range is a range that wholly encompasses a location. This means that the concept of a covering range can be applied to any location, whether it be a node, a point, or a range. Basically, a covering range is the smallest possible range covering a given location. In detail, this is defined as follows:

- For a range location, the covering range is identical to the range.
- For an attribute or namespace location (both node locations), the container node of the start point and end point of the covering range is the attribute or namespace location; the index of the startpoint of the covering range is 0; and the index of the endpoint of the covering range is the length of the string value of the attribute or namespace location.
- For the root location (a node location), the container node of the startpoint and endpoint of the covering range is the root node; the index of the start point of the covering range is 0; and the index of the endpoint of the covering range is the number of children of the root location.
- For a point location, the start- and endpoints of the covering range are the point itself.
- For any other kind of location, the container node of the startpoint and endpoint of the covering range is the parent of the location; the index of the startpoint of the covering range is the number of preceding sibling nodes of the location; and the index of the endpoint is one greater than the index of the startpoint.

In summary, XPointer's extensions to XPath's data model include the extension of the concept of nodes and node sets to that of locations and location sets (with the location being a node, a point, or a range). XPointer also introduces the concept of a covering range to support mapping of locations to ranges. For each location, there is a well-defined covering range.

In addition to these extensions of XPath's data model, XPointer also extends the concept of *document order,* as introduced in section 5.1. In general, the concept is extended not only to arrange nodes in a well-defined order but also to include point and range locations. Figure 6.2 shows how the locations of a document fragment are arranged in XPointer's document order. For defining rules regarding how to determine document order, XPointer introduces the concept of an *immediately preceding node* and then uses it to define how every possible combination of the relevant location types (i.e., nodes, points, and ranges) have to be compared to establish the document order of these locations.

6.1.2 XPointer Data Model Examples

To make these abstract definitions more understandable, we give some examples of how the concepts of locations, points, ranges, and nodes relate to each other. Our scenario is a browser that allows a user to create XPointers by selecting content with a mouse and then using a menu option for generating an XPointer identifying that content.

Marking a Point Within a Document

The simplest use is to mark a point within the document and generate an XPointer for this point (e.g., for creating an XPointer that is attached to an e-mail saying "please insert your text here"). To do this, the browser generates an XPointer that is a point. Depending on where the point has been selected, it is either a node point (if it has been marked within the root node or an element node) or a character point (in all other cases). Depending on how the browser was implemented, different XPointers that refer to the same point could be created. Consider the case where a user selects the point before the first character of a paragraph. Depending on the browser's implementation, this could result in the following:

- a node point into the paragraph's parent element node, identifying the point before the paragraph child,
- a node point with index zero within the paragraph's element node, or
- a character point with index zero within the text node of the paragraph's text.

All these cases make sense. In the first, the point could be used to insert elements before the paragraph element. In the second, the point could be used to insert elements into the paragraph before the first character; while in the third, it could be used to insert characters before the first character within the paragraph's text. It is entirely dependent on the implementation and application how these cases should be treated. Indeed, one possibility would be for the browser to present the user with a choice as to which is the appropriate XPointer.[11]

[11]Further complicating things, the browser could base its decision on the document's schema (e.g., the DTD) if available, which could be used to determine whether the paragraph element may have both text and children elements or only text.

Selecting Text Within One Node

Selecting text within one node is the equivalent of selecting two points,[12] the start and the end point of the selection, that lie inside the same node (e.g., the same element, attribute, or comment). From the user's point of view, it may not be apparent whether or not the two points are located within the same node (e.g., if two elements are not visibly separated by any formatting); but for the browser, this is easy to determine. Based on the selection of the user, the browser generates an XPointer defining a range between the selected start and end points. Since in this case the selected text lies within one node, the two points can easily be used to construct an XPointer range, which spans between the two points.

Selecting Text That Spans a Number of Nodes

If the start and the end point lie in different nodes, then the range defined by the two points also spans multiple nodes. According to the range constraints defined by XPointer, this is allowed only if the containing nodes of the startpoint and the endpoint are element, text, or root nodes (in all other cases, the start and endpoint must lie within the same node). A possible selection that spans a number of nodes in the case of an XHTML-like paragraph may select text that is all inside the paragraph but still spans multiple nodes. This may be because in between there are other element nodes (e.g., nodes representing emphasized text or hyperlinks). This situation effectively places the start and the endpoints in different text nodes (making them both children of the same paragraph node).

To further generalize this scenario, the start point and the end point may occur in entirely different subtrees of the document tree, for example, within a paragraph of the first chapter and in a table cell of the third chapter. This still would represent a valid range, spanning from the character point in the text node representing the first chapter's paragraph to the node point in the table row node directly after the selected table cell.

Making Multiple Selections

Even though today's user interfaces in most cases do not support this type of interaction, it would be perfectly reasonable to implement an interface that allows multiple non-contiguous selections. This could be used to create an XPointer that references multiple ranges within the same resource (e.g., attached to an e-mail saying "what do you think about these three statements?"). In a case such as this, it is mainly a question of the design of the user interface as to how multiple selections could be implemented in

[12]In XPointer, the startpoint of a range must occur before the endpoint in document order; so if the user makes the selection by first selecting the endpoint, the browser must recognize this and act accordingly.

a user-friendly way. From XPointer's point of view, the multiple selections could be easily combined to yield a single location set.

The preceding scenarios illustrate some typical cases in which XPointer concepts are relevant.[13] So far we have not discussed how exactly the hypothetical browser maps the concepts to actual XPointers, and as a first step toward resolving this issue, we first have to discuss the possible forms of XPointers.

6.2 XPOINTER FORMS

An XPointer is an identification of a part of an XML resource and is mainly intended to be used as a fragment identifier in a URI reference. As such, it must somehow fit into the framework defined for URIs as described in section 3.2. Basically, it must be represented as a printable string that can be used within URI references and exchanged in the same way URIs are exchanged. (To this end, XPointers must obey a number of character-escaping rules, which are not discussed here but are described in detail in section 6.4.1).

XPointer distinguishes three different forms as follows:

```
[1] XPointer                ::= Name
                              | ChildSeq
                              | FullXPtr
```

The first one (a *bare name*) is defined to act mainly as a very concise form and to be backwards compatible with HTML fragment identifiers. This first form is described in section 6.2.1. A second form (a *child sequence*) still uses an abbreviated syntax but allows more flexibility than the first form. It is described in section 6.2.2. Finally, the full form of XPointer is discussed in section 6.2.3. It is the most complex and most powerful XPointer form.

6.2.1 Bare Names

The simplest form of an XPointer is a *bare name*. Basically, it consists of the same name as that provided for the argument of a location step using the `id` function. This function returns the element that has an attribute

[13]Note that our examples assume that the presentation of a document in a browser can be used to identify points in the underlying source document. However, in the case of nontrivial presentations (such as when complex XSL style sheets that map single sections of an XML document to multiple presentation artifacts are used), this correspondence between selections in the presentation and the underlying document may become rather complicated. How this issue could and should be resolved in a general way is at the time of writing still the subject of discussions within the W3C's standardization process for XPointer.

of type ID[14] with the argument's value, so an XPointer bare name does exactly the same thing as an HTML fragment identifier.[15] As an example, the URI reference `http://www.w3.org/TR/#xptr` uses a bare name[16] and may consequently be interpreted as pointing to the element that has an attribute of type ID with the value `xptr`.

There are two reasons XPointer bare names are supported, despite their very limited functionality:

- For reasons of backwards compatibility with HTML and as an easy migration to XHTML without the need to rewrite all fragment identifiers when converting a resource from HTML to XHTML. Otherwise, HTML fragment identifiers become invalid if an HTML resource is converted to XHTML.
- To encourage the use of IDs, which are the most robust mechanism for pointing into XML documents.

XPointer bare names are very easy to understand and use, but they are limited in two ways.

The first limitation is that they may point only to element nodes, because only elements may be identified via ID attributes. The second limitation is that the element to which an XPointer should point must have an attribute of type ID, otherwise it is impossible to point to this element using a bare name. (If the document is under the control of the creator of the XPointer, then an ID may be created by modifying the document, but this is not always an option.)

As an intermediate form of XPointers (between the bare form and full XPointers), we have the child sequence. This form of XPointer, discussed in the next section, is also limited to pointing to element nodes, but it—in contrast to bare names—may point to element nodes that do not carry an attribute of type ID.

6.2.2 Child Sequences

A child sequence is a form of XPointer that selects an element node by navigating through a document's element tree. It resembles an XPath location

[14]Note that it is necessary to interpret the DTD to evaluate the type of an attribute.

[15]Except that in HTML, IDs can also be defined using an `<a>` element with a `name` attribute, which, although discouraged, is still widely used because of the lack of browser support for the newer identification mechanism using the `id` attribute defined for virtually every HTML element.

[16]To be more precise, this URI reference's fragment identifier may be regarded only as a bare name XPointer if the W3C's Web server reports an XML-based MIME type for the resource. If it reports an HTML MIME type, as it does at the time of writing, the fragment identifier must be interpreted as an HTML fragment identifier.

path in that it uses a path notation, but it is much more limited in that it can select only elements and uses only the child axis. The syntax for a child sequence is as follows:

```
[2] ChildSeq                 ::= Name? ('/' [1-9] [0-9]* )+
```

An example of a child sequence identifying the same element as the `http://www.w3.org/TR/#xptr` bare name would be `http://www.w3.org/TR/#/1/2/17/15/1/1/1`. This is nothing more than a navigation path through the element tree of the XHTML page based on child sequences. It could be spelled out as follows:

- Select the first child of the document root, which is the `<html>` element.
- Select the second child of the `<html>` element, which is the `<body>` element.
- Select the seventeenth child of the `<body>` element, which is a `<dl>` element.
- Select the fifteenth child of the `<dl>` element, which is a `<dt>` element.
- Select the first child of the `<dt>` element, which is a `<b>` element.
- Select the first child of the `<b>` element, which is an `<i>` element.
- Select the first child of the `<i>` element, which is an `<a>` element.

This last `<a>` element is the one carrying the `xptr` ID, which is why the bare name XPointer and the child sequence XPointer select the same element. In this case, we have selected something that is also accessible through an ID. However, what is interesting about the child sequence mechanism is that it would still be possible to select the element even if was not identified through an ID.

Child sequences have a severe disadvantage in that they are very sensitive to document modifications. It is very unlikely that the example just presented will still work after the W3C's page has been modified since the modification is likely to involve inserting or deleting elements preceding the one we want to select, resulting in breaking the intention of the child sequence.[17]

Since child sequences are easy to use but also easy to break, there is a second form of child sequence that uses an element identified by an ID

[17] We discuss more about the persistence of XPointers in section 6.4.4. Here it is interesting to note that the child sequence would probably still identify an element after the page has been modified, but most likely not identify the element that we want to identify.

as the starting point rather than using the document root. In this case, the XPointer starts with a bare name but then continues with a child sequence,[18] navigating the document tree starting from the element with the given ID. An example of this kind of child sequence would be `http://www.w3.org/TR/#last-call-list/15/1/1/1`. This would work if the W3C had made the `<dl>` element listing the document in "last call" status accessible through a `last-call-list` ID (which it has not done). This kind of child sequence would make the XPointer more robust to modifications of the document because only changes within the `<dl>` element could possibly break the XPointer.

6.2.3 Full XPointers

Both bare names and child sequences are rather limited in their expressiveness, insofar as they can select only element nodes and also in the way they select these nodes. Therefore, in many cases it is necessary to use more complex XPointers (the so-called full XPointer), which provide a much more flexible way of addressing into a document than bare names or child sequences do. Syntactically, a full XPointer is defined as follows:

```
 [3] FullXPtr                  ::= XPtrPart (S? XPtrPart)*
 [4] XPtrPart                  ::= 'xpointer' '(' XPtrExpr ')'
                                 | 'xmlns' '(' XPtrNsDecl? ')'
                                 | Scheme '(' SchemeSpecificExpr ')'
 [5] Scheme                    ::= NCName
 [6] SchemeSpecificExpr        ::= StringWithBalancedParens
 [7] StringWithBalancedParens  ::= [^()]* ('(' StringWithBalancedParens ')' [^()]*)*
 [8] XPtrExpr                  ::= Expr
 [9] XPtrNsDecl                ::= NCName S? '=' S? XPtrNsURI
[10] XPtrNsURI                 ::= Char*
```

As an example, now consider the W3C technical reports page. W3C has used IDs to mark individual sections (which are visible as headings and subheadings in Figure 6.1); but unfortunately in XHTML the actual contents of a section are not contained in any specific element. Instead they simply follow a sectioning element, such as the `<h3>` element used for the "Working Drafts in Last Call" section heading. Consequently, using only child sequences, it is impossible to point to the entry for the XPointer working draft entry. This is because the XPointer entry is part of the `<dl>` list following the "Working Drafts in Last Call" heading. So if we want to

[18]This is possible because XML rules state that name tokens, the syntax used for IDs, may not contain slash characters, which are used in defining child sequences.

point to the entry for XPointer starting from the heading with the ID, we have to use a full XPointer as follows:

```
http://www.w3.org/TR/#xpointer(id('last-call')/following::dl[1]/dt[7]//a)
```

Here we mainly exploit the fact that the "Working Drafts in Last Call" heading is identified by the `last-call` ID. The XPointer part of the URI reference could be interpreted as follows (commenting on the individual location steps):

- Select the element having the `last-call` ID. (This is the `<h3>` element with the "Working Drafts in Last Call" content.) Here we use XPath's `id` function as described in section 5.4.4.
- Select the first definition list (represented by a `<dl>` element) following the `<h3>` element. (For an explanation of the XPath `following` axis, refer to section 5.2.2.)
- Select the seventh definition term of this list (i.e., the seventh `<dt>` child of the `<dl>` element).
- Select the `<a>` element, which is (directly or indirectly) contained in the `<dt>` element.

In this example, we see that a normal XPath can be used within an XPointer. Apart from a few exceptions, XPointer allows unlimited use of XPath's expressiveness. Therefore, it is necessary to know XPath if we are to create anything more than trivial XPointers (i.e., bare names and child sequences).

Looking at the example just given, note the presence of a special keyword, `xpointer`, which, according to rule 4 from the syntactic definition, is the specification of a *scheme*. XPointer schemes are an important mechanism for building robust XPointers. According to rule 3 of the syntax, full XPointers use schemes to define any number of scheme-specific parts. XPointer specifies that scheme parts must be evaluated left to right, so the interpretation of the scheme parts is well defined. Currently, there are only two schemes defined for XPointer (as indicated by rule 4):

- `xpointer` – This is by far the most widely used scheme, and it indicates that the expression that follows (contained in parentheses) is an XPointer expression. This scheme has been used in the example just given.
- `xmlns` – This second scheme is for the initialization of namespaces. In XPath (and, therefore, in XPointer), an expression may contain a qualified name (i.e., a name containing a namespace prefix and a local name). In order to make the interpretation of qualified names possible,

there must be a mechanism for the initialization of namespaces (i.e., for the assignment of a namespace URI to a namespace prefix). This scheme will be discussed in detail in section 6.4.2.

While these two schemes are the only ones defined in the XPointer specification, it is possible for applications to define their own schemes, which can then be used to access subresources in an application-specific way. This would, however, make the XPointer unusable for any application not supporting the application-specific extension. It is useful to note that the XPointer specification states that interpretation proceeds from left to right and stops once a scheme part that can be successfully interpreted as a locator into the resource has been identified. Consequently, it is possible to concatenate multiple scheme parts, and if the interpretation of a scheme part fails (a so-called *subresource error*[19]), then the interpretation of the XPointer continues with the next scheme part. A simple example for this is the following URI reference:

```
http://www.w3.org/TR/#xpointer(id('xptr'))
    xpointer(/*[1]/*[2]/*[17]/*[15]/*[1]/*[1]/*[1])
```

In this case, the XPointer consists of two scheme parts, both being of the `xpointer` scheme. If an element with the ID `xptr` is present in the resource, then the XPointer will locate it.[20] Otherwise, the first scheme part results in a subresource error, and the second part then successfully locates the resource by simply counting element children.[21] Given our example used so far, this XPointer would continue to work even if the ID `xptr` was removed from the document. However, the XPointer would still break if the ID was removed and the resource's structure was changed in such a way that the "child sequence" no longer identified the element we want to locate.[22]

[19]XPointer defines three error types: *syntax errors* for XPointers being syntactically invalid, *resource errors* for XPointers pointing into nonexistent or non-XML resources, and *subresource errors*. Note that XPath allows empty node sets as results and does not regard this situation as an error. However, because XPointer is intended as a specification of document locations, an empty result is an error.

[20]Here it can be seen that a bare name XPointer used in a URI reference such as `...#name` is identical to the full form `...#xpointer(id('name'))`.

[21]In this case, it can be seen that a child sequence XPointer in a URI reference such as `...#/a/b/c` is identical to the full form `...#xpointer(/*[a]/*[b]/*[c])`. (In this case, the abbreviation mechanisms of XPath are used, otherwise each step would look like `child::*[position()=a]`, which makes explicit the child sequence positioning.)

[22]It should be noted that given the modification frequency of W3C's technical reports page, the structure of this page is likely to change at least once per week, even though some subtrees (such as the Recommendations section) may remain stable for months.

Formally, as already mentioned, the XPointer specification states that evaluation of XPointer scheme parts must be left to right but also that evaluation of scheme parts must continue if one of the following conditions is met while evaluating a scheme part:

- The scheme is unknown to the application evaluating the XPointer.
- The scheme is not applicable to the media type of the resource.
- The scheme does not locate any subresource present in the resource.[23] This applies to XPointer expressions evaluating to an empty location set (a subresource error).
- If the scheme being interpreted is `xpointer`, the following applies:
  - The string argument in a `string-range` function is not found in the string-value of the location, or the third or fourth argument of the function indicates a string that is beyond the beginning or end of the document.
  - The point returned by the `start-point` function is of type attribute or namespace.

This set of rules for the evaluation of XPointer scheme parts makes it possible to create XPointers with built-in "fault tolerance" (through providing fall-back scheme parts to be used if the original scheme part does not work anymore). In section 6.4.3 we will discuss some interesting examples.

6.3 FUNCTIONS

XPointer is built on top of XPath and extends XPath in several ways. At the most fundamental level, it extends XPath's data model, and this aspect has been covered in section 6.1.1. However, in order to fully exploit this extended data model, XPointer also extends the list of functions provided by XPath. XPath's functions have been described in section 5.4, with an overview provided in Table 5.4. XPointer significantly extends this list of functions. The additional functions provided by XPointer are summarized in Table 6.1.

Before covering the functions in detail, we should note that when XPath was developed it was assumed that the XPath functions would be extended by other specifications. (Indeed, as we have mentioned previously, XPath

[23]Note that an XPointer part that uses the `xmlns` scheme never returns a subresource and thus always fails. However, its evaluation has a potential effect on XPointer parts to its right (see section 6.4.2 for more information).

Table 6.1 Overview of XPointer Functions

Function name	Result type	Arguments	Page
`end-point`	`location-set`	`location-set`	157
`here`	`location-set`	n/a	158
`origin`	`location-set`	n/a	158
`range`	`location-set`	`location-set`	158
`range-inside`	`location-set`	`location-set`	158
`range-to`	`location-set`	`location-set`	159
`start-point`	`location-set`	`location-set`	160
`string-range`	`location-set`	`location-set, string, number?, number?`	160

is more intended as a foundation for other specifications than as a stand-alone standard.) Consequently, the extension of XPath's set of functions is perfectly legal and well defined within the XPath model.

One particularly useful observation about the XPointer functions is that they all accept arguments and produce results using the extended XPointer data model. Furthermore, from the function names it can be concluded that the majority of functions are used for defining ranges. This is not surprising given that ranges are XPointer's most important extension of the XPath data model and are not supported at all by XPath's functions. In the following list, we cover all XPointer functions (supplementing the core XPath functions) in detail:

- `end-point` – *Returns the end point of a location.*
  Signature: `location-set end-point(location-set)`
  The `end-point` function accepts a location set as its argument. For each of the locations in this location set, the function adds the end point of the location to the resulting location set, according to the following rules (with x being a location in the argument location set):
  - If x is of type point, the resulting point is x.
  - If x is of type range, the resulting point is the end point of x.
  - If x is of type root or element, the container node of the resulting point is x, and the index is the number of children of x.
  - If x is of type text, comment, or processing instruction, the container node of the resulting point is x, and the index is the length of the string-value of x.
  - If x is of type attribute or namespace, the XPointer part in which the function appears fails.

Thus, the `end-point` function can be used to locate the end point of any location. The complementary function of the `end-point` function is the `start-point` function.

- `here` – *Returns the context of the XPointer.*
  Signature: `location-set here()`
  The `here` function makes it possible to create XPointers relative to the context in which they appear. Since this makes sense only for XPointers within XML resources, an XPointer part containing the `here` function always fails if the XPointer does not appear inside an XML resource.[24] If the containing resource is XML, the `here` function returns a location set with one member: If the XPointer being evaluated appears in a text node inside an element node, the location returned is the element node. Otherwise, the location returned is the node that directly contains the XPointer being evaluated.
- `origin` – *Returns the context of the traversal initiation.*
  Signature: `location-set origin()`
  This function makes sense only in a context where XPointers are used within linking constructs (such as those provided by XLink) and where the processing model is such that the XPointer evaluation is initiated by the traversal of links. If this is the case, the `origin` function returns a location set with one member, which locates the element from which a user or program initiated traversal of the link.
  If the `origin` function is used in a URI reference where a URI is also provided and identifies a containing resource different from the resource from which traversal was initiated, the result is a resource error. It is also a resource error to use the `origin` function in a context where traversal is not occurring.
- `range` – *Returns covering ranges of locations.*
  Signature: `location-set range(location-set)`
  The `range` function returns the covering ranges of all locations in the argument location set. Thus, the result location set of the `range` function always contains a set of range locations, and this set contains, at most, as many range locations as there are locations in the argument location set.[25]
- `range-inside` – *Returns ranges covering the contents of locations.*
  Signature: `location-set range-inside(location-set)`
  The `range-inside` function is similar to the `range` function in that it

[24]One possible scenario would be an XPointer being typed into the address bar of a browser, which would not have an XML document as context.

[25]It contains fewer range locations if at least some of the argument locations have the same covering ranges.

also returns ranges covering the locations in the argument location set. However, the `range-inside` function does not return the covering ranges for the locations in the argument location set but instead returns ranges covering the contents of these locations. The following rules are used to construct the result location set, based on the type of each location in the argument location set:

– For range locations, the location (i.e., the range) is added to the result location set.
– For point locations, the location (i.e., the point) is added to the result location set. Consequently, the result location set of the `range-inside` function can contain range and point locations.
– For node locations, the location (i.e., the node) is used as the container node of the start and end points of the range location to be added to the resulting location set. The index of the start point of the range is zero. If the end point is a character point, then its index is the length of the string value of the argument node location; otherwise it is the number of children of the argument node location.

This definition of the `range-inside` function makes sure that only the contents of locations are added to the result location set. For example, if the argument location is an element, then the `range-inside` function returns the contents of this element as the result (in contrast, the `range` function would return the element itself).

- `range-to` – *Returns range from context location to argument location.*
Signature: `location-set range-to(location-set)`
This function has a special position among the other functions in that it requires a change of the XPath syntax as described in section 5.2.[26] In XPointer, rule 4 of XPath's syntax of location paths is changed from

```
[4] Step               ::= AxisSpecifier NodeTest Predicate*
                         | AbbreviatedStep
```

to the following form:

```
[4xptr] Step          ::= AxisSpecifier NodeTest Predicate*
                        | AbbreviatedStep
                        | 'range-to' '(' Expr ')' Predicate*
```

[26]This way of incorporating the `range-to` functionality into XPointer is not very elegant; and in order to avoid similar situations in the future, the XPointer specifications states: "This change is a single exception for the `range-to` function. It is not a generic change and is not extensible to other functions. The modified production expresses that a range computation must be made for each of the nodes in the current node list" [DeRose+ 01a].

This modification of the syntax makes it possible to use the `range-to` function directly as a step of a location path (instead of the situation with other functions, which may be used only within predicates or other expressions). The `range-to` function operates on the context provided by the previous step and produces the context for the following step.

For each location in the context, the `range-to` function returns a range. The start of this range is the start point of the context location, and the end of the range is the end point of the location found by evaluating the expression argument with respect to that context location. Thus, if the context is a location set with more than one location, then for each of these locations, the `range-to` function's argument is evaluated with respect to the location. The result of evaluating the `range-to` function then is the union of all ranges that are the results of these evaluations.

- `start-point` – *Returns the start point of a location.*
  Signature: `location-set start-point(location-set)`
  The `start-point` function accepts a location set as its argument. For each of the locations in this location set, the function adds the start point of the location to the resulting location set, according to the following rules (with x being a location in the argument location set):
  - If x is of type point, the start point is x.
  - If x is of type range, the start point is the start point of x.
  - If x is of type root, element, text, comment, or processing instruction, the container node of the start point is x, and the index is 0.
  - If x is of type attribute or namespace, the XPointer part in which the function appears fails.

  Thus, the `start-point` function can be used to locate the start point of any location. The complementary function of the `start-point` function is the `end-point` function.
- `string-range` – *Matches strings in a location set.*
  Signature: `location-set string-range(location-set, string, number?, number?)`
  This is one of the most important (and complex) functions provided by XPointer. In many cases, it is necessary not only to use the structure provided by XML (such as elements, attributes, or processing instructions) for identifying resource fragments but also to be able to identify fragments that are text-based. Basically, the `string-range` function enables the identification of strings (or sets of strings) as ranges (or sets of ranges).

For each location in the argument location set, the location's string value (see section 5.1 for the definition of the string value) is searched for the given string.[27] Each non-overlapping match of this string is then added (as a range location) to the resultant location set. If no matching string exists, then the XPointer part (within which the `string-range` function appears) fails.

The optional third and fourth arguments can be used to control the range, which is added to the resulting location set. The third argument specifies from which point, relative to the start of the matched string, the result should be taken. The default value for the third argument is 1, which means the result should be from before the first character matching the search string. The fourth argument specifies the length of the range to be added. The default is the range that extends to the end of the matched string.

These functions can be used to compose XPointers. It should also be noted that most XPath functions can also be used within XPointers. In particular, it can be observed (most easily from looking at Table 6.1) that XPointer's functions are mainly concerned with supporting the concept of locations, which are a construct introduced by XPointer mainly for the purpose of including ranges in the data model.

6.4 USING XPOINTERS

So far we have discussed the specifics of XPointer as a way of identifying resource fragments. In this section, we talk about some of the issues that arise when using XPointers. One of the obvious problems when creating and using XPointers in an environment that is essentially character-based is the issue of character escaping (described in section 6.4.1). Another topic of a similar nature is the question of how to use XPointers with XML Namespaces (discussed in section 6.4.2).

XPointers identify resource fragments by describing ways for locating them inside the resources. As pointed out earlier, this can be done in an endless variety of ways, so composing XPointers is not a mechanical process. Rather, it requires some intelligence in order to compose "good" XPointers. Since XPointer is a new technology, there is not much implementation

[27]The string given as `string-range`'s argument is matched literally, which means it is case-sensitive, no regular expressions of any type are possible, and the only normalization that is done is the whitespace handling as defined by XML (which is, in particular, important for line ends). Sophisticated string matching (such as regular expressions) is not supported by XPointer and must be handled at the application level.

experience to build on; but nevertheless in section 6.4.3 we describe some guidelines for composing good XPointers. Finally, in section 6.4.4 we look into the question of what exactly *good* means. In many cases, it will have a lot to do with ensuring XPointer persistence.

6.4.1 XPointer Character Escaping

While composing XPointers is based on the way subresources within XML documents should be identified, they must also be coded in a way that makes it possible to exchange and interpret them unambiguously. XPointers use a character-based notation and are thus easy to compose and read. But several characters within XPointers have special meaning and must therefore be escaped, if they have to be embedded into XPointers. Because of the different standards involved when actually using an XPointer, escaping mechanisms occur on different levels, as follows:

- *XPointer escaping rules.* The XPointer specification defines escaping rules for some special characters. Most importantly, parentheses in XPointer must be balanced. This is because XPointer is built on the assumption that the end of syntactic constructs using parentheses can be found by identifying the balanced parenthesis. Consequently, unbalanced parentheses in XPointers must be escaped, and this is done by prefixing them with the circumflex character, "`^`". This makes it necessary to also escape the circumflex character, which is done by escaping it with itself (i.e., the literal circumflex character within an XPointer is written as "`^^`").
- *XML escaping rules.* Very often XPointers will be used within XML documents, and in this case XML's rules for escaping XML special characters must be observed. This means that any characters not representable in the character encoding of the XML document (as well as any characters relevant for XML markup) must be written as character references or as predefined entity references.
- *URI escaping rules.* URI references must adhere to the syntactic rules defined by RFC 2396 [Berners-Lee+ 98], which allows only a limited set of characters. All other characters must be represented using the URI escape mechanism, which represents these characters by a percent sign, "`%`", followed by two hexadecimal digits.

These character-escaping rules in many cases must be combined when XPointers are used in an XML-based environment. Consequently, character escaping can become quite complicated. The XPointer specification gives

Table 6.2 XPointer Character Escaping (Example 1)

Level	**Example**
Initial	`xpointer(string-range(//P,"a little hat ^"))`
XPointer	`xpointer(string-range(//P,"a little hat ^^"))`
XML	`xpointer(string-range(//P,&quot;a little hat ^^&quot;))`
URI	`xpointer(string-range(//P,%22a%20little%20hat%20%5E%5E%22))`

Table 6.3 XPointer Character Escaping (Example 2)

Level	**Example**
Initial and XPointer	`xpointer(id('résumé'))`
XML	`xpointer(id('r&#xE9;sum&#xE9;'))`
URI	`xpointer(id('r%C3%A9sum%C3%A9'))`

some examples of how the different escaping mechanisms affect a given XPointer (reproduced in Tables 6.2 and 6.3).

In the first example (Table 6.2), it is interesting to see that even trivial things such as space characters must be escaped in the URI encoding because spaces are not allowed to appear literally within URIs. In the XML encoding, the double quotes must be escaped only if the XPointer appears within an XML attribute that is delimited with double quotes.

The second example (shown in Table 6.3) shows how to deal with non-ASCII characters. Because XPointer is based on Unicode, the accented letter appears both in the initial and the XPointer form. Based on the assumption that the XML document supports only ASCII, the accented letter must be represented by a character reference to its Unicode code point [Unicode 00]. In the URI encoding, however, the accented character has first to be encoded in UTF-8 [Yergeau 98] before the resulting byte sequence is escaped, so the result looks quite different from the XML escaping.

6.4.2 XPointers and Namespaces

In section 6.2.3, we discussed how XPointer defines the concept of schemes (in fact, each full XPointer is nothing more than a sequence of scheme-specific parts) and that currently only the `xpointer` and the `xmlns` schemes are specified. XPath (and thus, XPointer) makes it possible to use qualified names that have a namespace prefix and a local part. (For a discussion of qualified names and XML Namespaces in general, see section 4.2.) In

XML documents, the namespace prefix can easily be interpreted because, in order for the qualified name to be valid, there must be a namespace declaration associating that prefix with a namespace URI somewhere on an ancestor element.[28] This is no problem since the qualified names in XML are embedded into the context provided by the XML document (in particular, the namespace declarations within this document). An XPointer, however, does not have such a context because it may be used outside any document, simply as part of a URI reference. Consequently, there must be a way to establish the context of namespace declarations for XPointers.

XPointer defines the `xmlns` scheme for declaring namespaces. This is, in a way, very similar to namespace declarations in XML documents. Each `xmlns` scheme part associates one namespace prefix with a namespace URI. However, the syntax is slightly different from the one used in XML. The syntax is defined in rules 9 and 10 of the standard, as shown in section 6.2.3 (and repeated here):

```
 [4] XPtrPart              ::= 'xpointer' '(' XPtrExpr ')'
                             | 'xmlns' '(' XPtrNsDecl? ')'
                             | Scheme '(' SchemeSpecificExpr ')'
 [9] XPtrNsDecl            ::= NCName S? '=' S? XPtrNsURI
[10] XPtrNsURI             ::= Char*
```

Thus, whenever an XPointer is used that contains qualified names, it has to contain `xmlns` scheme parts for declaring the prefixes being used in the qualified names, as shown in the following example:

```
...#xmlns(html=http://www.w3.org/1999/xhtml)xpointer(//html:h3[9])
```

It is important to note that the prefix used in the XPointer and the prefix used in the resource need not be the same in order for the XPointer to match. The important part in this case is the namespace URI, so it is necessary only that the URI in the XPointer `xmlns` scheme part and the URI in the XML document (i.e., the namespace declaration using the `xmlns` attribute) are the same.

If two `xmlns` scheme parts within one XPointer declare the same prefix, then the second (i.e., right) declaration overrides the first (i.e., left) one. However, because evaluation of XPointer scheme parts is done stepwise from left to right, an `xpointer` scheme part that appears between the two `xmlns` scheme parts declaring the same prefix will be interpreted using the first declaration.

[28]This is with the exception of the `xml` prefix, which is always bound to the namespace URI of the XML Namespaces standard.

6.4.3 How to Compose XPointers

In section 5.5 we described in detail how to use XPath. The same principles apply to XPointer, particularly the key points of "being as specific as possible" and "filtering as early as possible." However, it is important to see the difference in possible application scenarios:

- *XPath and XSLT.* Today, the most frequently used application of XPath is in XSLT. In XSLT, XPaths may be evaluated very often during the processing of a style sheet, so it is important to keep an eye on the efficiency of the XPaths being used. Furthermore, the XSLT author often also controls the XML document (as well as the schema behind it), which makes it easier to compose XPaths that are not compromised by modifications to documents or even the schema.
- *XPath and XPointer.* In XPointer, however, XPath is often used for identifying fragments in resources not under the control of the XPointer's author. On the other hand, XPointers are usually evaluated only once (when locating the fragment within the resource), so efficiency is not a significant issue. Robustness, on the other hand, is very important since the XPointer should continue to work even if the resource that it points into changes.

Consequently, there is a difference between using XPath in the context of XSLT and using it in the context of XPointer. In general, the most important aspect of composing XPointers is robustness, which therefore is discussed separately in section 6.4.4.

As pointed out already, there are countless ways for each given fragment to be identified by an XPointer. We have already demonstrated this with the example of W3C's technical reports Web page at the start of this chapter. Continuing this discussion, we could add that, if, for example, the heading was not identified by the `last-call` ID, it would be possible to locate the heading based on its content,[29] as shown in the following URI:

```
http://www.w3.org/TR#xpointer(//h3[contains(string(.),
  'Working%20Drafts%20in%20Last%20Call')/following::dl[1]/dt[7]//a)
```

However, it would not make much sense to list a huge number of possible XPointers identifying the same resource because this list would never be exhaustive. Furthermore, without knowing the schemas behind the resources and the characteristics of how they are modified, it is hard to actually rate the many variants qualitatively.

[29]Because the XPointer is used in a URI reference here, the space characters in the search string must be URI-encoded as `%20` (as described in section 6.4.1).

The main point is that anybody involved in the creation of XPointers (either manually or programmatically) should be aware that this is not a strict science but more of an art form. In particular, any software generating XPointers should be carefully designed in order to generate good XPointers, and doing this is a non-trivial task. The quality criteria depend on the application domain and on how much knowledge there is available about the resources being used. In particular, one criterion that will very often be highly ranked is the persistence of XPointers.

6.4.4 Persistence

As discussed in the introduction to this chapter, the persistence of an XPointer is a serious issue (see also the discussion in section 3.3.2). Even if the resource addressed by a URI is still available, it may have changed, and the XPointer may not work any more, or it may not work as expected. In order to construct robust XPointers (i.e., XPointers that are tolerant against modifications of the resource), it is necessary to follow some guidelines. These guidelines, however, depend on how much is known about the resources being used.

As an example, it is fairly certain that the W3C will keep its overall structure of the technical reports page (see Figure 6.1), and that additions to the individual sections (e.g., "Working Drafts in Last Call") will always be at the start of the section. It would therefore be a reasonable idea to use a section's ID and to then start to count from the end of the list contained within the following:

```
http://www.w3.org/TR#xpointer(id('last-call')/
    following::dl[1]/dt[last()-4]//a)
```

While this approach could (and probably would) work within the "Recommendations" or "Notes" sections (where documents remain), it is less likely to work in the "Working Drafts in Last Call" section where working drafts may change status and may be deleted. By now it should be clear that constructing robust XPointers requires a good deal of knowledge about the resources, which may be impractical or too expensive to acquire.

Using IDs is always a good idea; and as long as there are IDs being used within the resource, it is a good idea to start with an ID and then navigate from there. But again, if the schema of a resource is unknown, then it is not really possible to find out which attributes are used as IDs. (However, it may be possible to make an educated guess, such as looking for attributes with unique values or attributes having the string `id` as part of their name.)

Besides all these worries about XPointers becoming invalid or incorrect because of resource modifications, it should always be remembered that even though XPointer is the W3C standard for XML fragment identification, it is not the only means of identifying fragments. If, for example, the validity of fragments is very important, then one alternative might be not only to generate XPointers but also to generate information for checking the XPointer's validity. One approach to this might be to use the modification time of the resource at the time the XPointer was generated, or even a digital fingerprint of the resource or subresource (using checksum algorithms such as MD5 [Rivest 92] or SHA [NIST 93]).

Information such as dates or checksums could easily be incorporated into XPointers themselves by using proprietary schemes, which, by definition, would be ignored by applications not knowing or supporting them. That way it would be possible to generate XPointers that would work on all platforms supporting XPointer and that would have the added benefit of being able to be tested for possible modifications of the fragment by platforms supporting the additional XPointer scheme.

6.5 FUTURE DEVELOPMENTS

Of all the W3C specifications relevant to XML linking, XPointer is the standard that has progressed most slowly. There are several reasons. One is that adoption of XPointer has been very slow, and at the time of writing there is only one implementation available. This is not sufficient to act as a catalyst for the W3C standardization process to continue. Some implementors are concerned that XPointer is too complex (in particular, its concept of ranges) and that this keeps vendors from supporting it. There have also been attempts to create profiles of XPointer (which exclude ranges and are therefore much easier to implement); but this would significantly reduce the functionality supported by the core standard, and so far the W3C working group has not agreed to this plan. Unfortunately, XPointer is one of the essential building blocks of the XML linking framework; but as long as the W3C standardization process remains stalled, there will be either little or no progress or some vendors will introduce proprietary solutions to the problem of addressing subresources.

Apart from these more political issues, XPointer is already becoming somewhat outdated by the continuing development of the standards on which it is based—most particularly, XPath. While XPointer is built on top of XPath 1.0, XPath 2.0 [Berglund+ 01], described in section 5.6, is already under development, and it is very possible that it will reach recommendation status earlier than XPointer. Because of the problems with

XPointer standardization, currently no attempts are being made to base XPointer on XPath 2.0.

It is our opinion that it is often better to have a mediocre standard than no standard at all. While XPath and XSLT have demonstrated that even non-perfect standards can provide many benefits (and the opportunity to improve them with their next release), XPointer's development is an example of how the lack of standardization of an essential component of a bigger framework (the XML linking technologies) can stall the development of a very interesting and promising set of technologies and applications.

6.6 CONCLUSIONS

XPointer is the official fragment identifier for XML documents. Using XPointer, it is possible not only to link complete resources but also to create links between parts of resources. XPointer is built on top of XPath, and it extends XPath's model with some new concepts. This chapter describes XPointer in detail and demonstrates to readers how to create their own XPointers. These XPointers may then be used in links, which are described in the following chapter.