WD-xptr-19980303

XML Pointer Language (XPointer)

World Wide Web Consortium Working Draft 03-March-1998

This version:
http://www.w3.org/TR/1998/WD-xptr-19980303
Previous version:
http://www.w3.org/TR/WD-xml-link-970731
Latest version:
http://www.w3.org/TR/WD-xptr
Editors:
Eve Maler (ArborText) <elm@arbortext.com>
Steve DeRose (Inso Corp. and Brown University) <sderose@eps.inso.com>

Status of this document

This is a W3C Working Draft for review by W3C members and other interested parties. It is a draft document 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". A list of current W3C working drafts can be found at http://www.w3.org/TR.

This work is part of the W3C XML Activity (for current status, see http://www.w3.org/MarkUp/XML/Activity). For information about the XLink language in which XPointer is expected to be used, see http://www.w3.org/TR/WD-xlink .

See http://www.w3.org/TR/NOTE-xlink-principles for additional background on the design principles informing XPointer.

Abstract

This document specifies constructs that support addressing into the internal structures of XML documents. In particular, it provides for specific reference to elements, character strings, and other parts of XML documents, whether or not they bear an explicit ID attribute.

XML Pointer Language (XPointer)

Version 1.0

Table of Contents

---

1. Introduction

This document specifies a language that supports addressing into the internal structures of XML documents. In particular, it provides for specific reference to elements, character strings, and other parts of XML documents, whether or not they bear an explicit ID attribute.

1.1 Language Design Goals

Following is a summary of the design principles governing XPointer:

1. XPointers shall address into XML documents.
2. XPointers shall be straightforwardly usable over the Internet.
3. XPointers shall be straightforwardly usable in URIs.
4. The XPointer design shall be prepared quickly.
5. The XPointer design shall be formal and concise.
6. The XPointer syntax shall be reasonably compact and human readable.
7. XPointers shall be optimized for usability.
8. XPointers must be feasible to implement.

1.2 Relationship to Existing Standards

Three standards have been especially influential:

- *HTML:* Has popularized an important location specifier type, the URL (now URI).
- *HyTime:*Defines location specifier types for all kinds of data.
- *Text Encoding Initiative Guidelines (TEI P3):* Provide a formal syntax for location specifiers for structured data, graphics, and other data.

Many other linking systems have also informed this design, especially Dexter, FRESS, MicroCosm, and InterMedia.

1.3 Terminology

The following basic terms apply in this document.

element tree
: An abstract representation of the relevant structure specified by the tags, attributes, and other markup constructs in an XML document.

link
: An explicit relationship between two or more data objects or portions of data objects.

linking element
: An element that asserts the existence and describes the characteristics of a link.

locator
: Data, provided as part of a link, which identifies a resource.

resource
: In the abstract sense, an addressable unit of information or service that is participating in a link. Examples include files, images, documents, programs, and query results. Concretely, anything reachable by the use of a locator in some linking element. Note that this term and its definition are taken from the basic specifications governing the World Wide Web.

sub-resource
: A portion of a resource, pointed to as the precise destination of a link. As one example, a link might specify that an entire document be retrieved and displayed, but that some specific part(s) of it is the specific linked data, to be treated in an application-appropriate manner such as indication by highlighting, scrolling, etc.

1.4 Notation

The formal grammar for locators is given using a simple Extended Backus-Naur Form (EBNF) location, as described in the XML specification.

2. XPointers in Locators

The locator for a resource is typically provided by means of a Uniform Resource Identifier, or URI. XPointers can be used as fragment identifiers in conjunction with the URI structure to specify a more precise sub-resource. Any fragment identifier that points into an XML resource must be an XPointer. However, for any locator in an XML resource that identifies a resource that is not an XML document (for example, an HTML or PDF document), XPointer does not constrain the syntax or semantics of the locator.

3. The XPointer Language

XPointers operate on the tree defined by the elements and other markup constructs of an XML document.

An **XPointer** consists of a series of location terms, each of which specifies a location, usually relative to the location specified by the prior location term. Each location term has a keyword (such as `id`, `child`, `ancestor`, and so on) and can have arguments such as an instance number, element type, or attribute. For example, the location term `child(2,CHAP)` refers to the second child element whose type is `CHAP`.

3.1 XPointer Structure

At the heart of the XPointer is the **location term**, the basic unit of addressing information. The combination of location terms in an XPointer has the effect of specifying a precise location.

XPointer

```
[1]  XPointer  ::= AbsTerm '.' OtherTerms
                 | AbsTerm
                 | OtherTerms
[2] OtherTerms ::= OtherTerm
                 | OtherTerm '.' OtherTerm
[3]  OtherTerm ::= RelTerm
                 | SpanTerm
                 | AttrTerm
                 | StringTerm
```

Many XPointers locate individual nodes in an element tree. However, some location terms can locate more complex sets of data. For example, a string match may locate only a portion of a node, and an XPointer containing the `span` location term (called a **spanning XPointer**) can reference sub-resources that do not constitute whole elements.

Note that the implementation of traversal to a resource is not constrained by this specification. In particular, handling a resource designated by a span is probably highly application-dependent. In a display-oriented application program, such traversal might simply highlight the designated characters; but a structure-oriented viewer might have no interest in sub-resources that are not complete nodes or subtrees. Note that a span cannot be treated as a set or list of elements, because spans may locate partial elements.

Location terms are classified into absolute terms, relative terms, span terms, attribute terms, and string data terms. An absolute term selects one or more elements or locations in an XML document without reference to any other sub-resource location. A relative or string data term specifes a location in terms of another location, called the **location source** . The location source is the entire resource if there are no preceding location terms; otherwise it is the location specified by the preceding term (which might be relative to a location term before that).

3.2 Absolute Location Terms

The keywords described in this section do not depend on the existence of a location source. They can be used to establish a location source or can serve as self-contained XPointers.

Absolute location terms

```
[4]  AbsTerm ::= 'root()' | 'origin()' | IdLoc | HTMLAddr
[5]    IdLoc ::= 'id(' Name ')'
[6] HTMLAddr ::= 'html(' SkipLit ')'
```

The empty parentheses after `root` and `origin` are for consistency with other keywords and to avoid ambiguous interpretation of an XPointer containing just the string "root" or "origin".

3.2.1 The root Keyword

If an XPointer begins with `root()`, the location source is the root element of the containing resource. If an XPointer omits any leading absolute location term (that is, it consists only of `OtherTerms`), it is assumed to have a leading `root()` absolute location term.

3.2.2 The origin Keyword

The `origin` keyword produces a meaningful location source for any following location terms only if the XPointer is being processed by application software in response to a request for traversal such as defined in the XLink specification. If an XPointer begins with `origin()`, the location source is the sub-resource from which the user initiated traversal rather than the default root element. This allows XPointers to select abstract items such as "the next chapter".

It is an error to use `origin()` in a locator where a URI is also provided and identifies a containing resource different from the resource from which traversal was initiated.

3.2.3 The id Keyword

If an XPointer begins with `id(Name)`, the location source is the element in the containing resource with an attribute having a declared type of `ID` and a value matching the given `Name`.

For example, the location term `id(a27)` chooses the necessarily unique element of the containing resource which has an attribute declared to be of type `ID` whose value is `a27`.

Note that if an XML document does not declare all attributes whose values are intended to serve as unique IDs, application software cannot reliably distinguish ID attributes from others with the same string value. Application software processing an XPointer must first attempt to locate an element with a declared ID attribute whose value matches that `Name` argument. If unable to do so, at user option the application software may locate any element having an attribute with the desired value.

3.2.4 The html Keyword

If an XPointer begins with `html(NAMEVALUE)`, the location source is the first element whose type is `A` and which has an attribute called `NAME` whose value is the same as the supplied `NAMEVALUE` . This is exactly the function performed by the "#" fragment identifier in the context of an HTML document.

3.3 Relative Location Terms

The keywords described in this section depend on the existence of a location source. If none is explicitly provided, the location source is the root element of the containing resource. These location terms provide facilities for navigating forward, backward, up, and down through the element tree. These location terms all accept the same list of arguments.

Relative location terms

```
[7] RelTerm ::= Keyword? Arguments
[8] Keyword ::= 'child'
                | 'descendant'
                | 'ancestor'
                | 'preceding'
                | 'following'
                | 'psibling'
                | 'fsibling'
```

Each of these keywords identifies a sequence of elements or other XML node types from which the resulting location source will be chosen. The arguments passed to the keyword determine which node types from that sequence are in fact chosen. Each keyword summarized here is described in detail in the following sections.

`child`
: Identifies direct child nodes of the location source.

`descendant`
: Identifies nodes appearing anywhere within the content of the location source.

`ancestor`

Identifies element nodes containing the location source.

preceding
Identifies nodes that appear before (preceding) the location source.

following
Identifies nodes that appear after (following) the location source.

psibling
Identifies sibling nodes (sharing their parent with the location source) that appear before (preceding) the location source.

fsibling
Identifies sibling nodes (sharing their parent with the location source) that appear after (following) the location source.

If the keyword is omitted, it is treated as equivalent to the immediately preceding keyword; the keyword must not be omitted from the first location term of any XPointer (including embedded ones). For example, the following two XPointers are equivalent:

```
child(2,SECTION).(1,SUBSECTION)
```

```
child(2,SECTION).child(1,SUBSECTION)
```

3.3.1 Relative Location Term Arguments

All relative location terms operate using the same set of potential arguments:

Relative location term arguments

```
[9] Arguments ::= '(' InstanceOrAll
                  (',' NodeType
                  (',' Attr ',' Val)*)? ')'
```

3.3.2 Selection by Instance Number

Elements and other node types can be selected by occurrence number:

Instance

```
[10] InstanceOrAll ::= 'all' | Instance
[11]       Instance ::= ('+' | '-')? [1-9] Digit*
```

For a positive instance number *n*, the *n*th of the candidate locations is identified. For a negative instance number, the candidate locations are counted from last to first (in a manner that is specific to each keyword). If the instance value `all` is given, then all the candidate locations are selected. Numbers that are out of range cause the XPointer to fail.

3.3.3 Selection by Node Type

XML sub-resources can be selected by specific node type as well as number:

Node type

```
[12] NodeType ::= Name
                | '#element'
                | '#pi'
                | '#comment'
                | '#text'
                | '#cdata'
                | '#all'
```

The node type may be specified by one of the following values:

Name
Selects a particular XML element type; only elements of the specified type will count as candidates. For example, the following identifies the 29th paragraph of the fourth sub-division of the third major division of the location source:

```
child(3,DIV1).child(4,DIV2).child(29,P)
```

The following XPointer selects the last `EXAMPLE` element in the document:

```
descendant(-1,EXAMPLE)
```

#element
Identifies XML elements. If no `NodeType` is specified, `#element` is the default. The following example identifies the fifth child element:

```
child(5)
```

#pi
Identifies XML processing instructions. This node type cannot satisfy any attribute constraints. The only location term that can meaningfully be used with a PI location source is `string`.

#comment
Identifies XML comments. This node type cannot satisfy any attribute constraints. The only location term that can meaningfully be used with a comment location source is `string`.

#text
Selects among text regions directly inside elements and CDATA sections. This node type cannot satisfy any attribute constraints. The only location term that can meaningfully be used with a text-region location source is `string`.

#cdata
Selects among text regions found inside CDATA sections. This node type cannot satisfy any attribute constraints. The only location term that can meaningfully be used with a CDATA-region location source is `string`.

#all
: Selects among nodes of all the above types. No node but an element can satisfy any attribute constraints, so if attribute constraints are provided, `#all` is effectively equivalent to `#element`.

Among the node types, elements can contain other types, but no other types can contain anything but strings. Thus, for example, `ancestor` location terms locate only element node types, and `descendant` location terms navigate downward through elements (not other node types) to reach the desired element or non-element node type.

Selection by a named element type when possible is strongly recommended; see "3.8 Link Persistence" for more information.

Consider the following example:

```
<!DOCTYPE SPEECH [
<!ELEMENT SPEECH (#PCDATA|SPEAKER|DIRECTION)*>
<!ATTLIST SPEECH
        ID      ID      #IMPLIED>
<!ELEMENT SPEAKER (#PCDATA)>
<!ELEMENT DIRECTION (#PCDATA)>
]>
<SPEECH ID="a27"><SPEAKER>Polonius</SPEAKER>
<DIRECTION>crossing downstage</DIRECTION>Fare you well,
my lord. <DIRECTION>To Ros.</DIRECTION>
You go to seek Lord Hamlet? There he is.</SPEECH>
```

The following XPointers select various sub-resources within this resource:

id(a27).child(2,DIRECTION)
: Selects the second "`DIRECTION`" element (whose content is "` To Ros.`").

id(a27).child(2,#element)
: Selects the second child element (that is, the first direction, whose content is "`crossing downstage`").

id(a27).child(2,#text)
: Selects the second text region , "`Fare you well, my lord.`" (The line break between the `SPEAKER` and `DIRECTION` elements is the first text region.)

3.3.4 Selection by Attribute

Candidate elements can be selected based on their attribute names and values. Note that non-element node types have no attributes, and so can never satisfy selection criteria that include attribute name or value specifications.

Attribute

```
[13] Attr ::= '*'          /* any attribute name */
             | Name
[14]  Val ::= '#IMPLIED' /* no value specified, no default */
             | '*'         /* any value, even defaulted */
             | Name
             | SkipLit  /* exact match */
```

The Attr and Val arguments are used to provide attribute names and values to use in selecting among candidate elements.

If specified within quotation marks, the attribute-value argument is case-sensitive; otherwise not.

Attribute names may be specified as "*" in location terms in the (unlikely) event that an attribute value constitutes a constraint regardless of what attribute name it is a value for.

For example, the following location term selects the first child of the location source for which the attribute TARGET has a value:

```
child(1,#element,TARGET,*)
```

The following XPointer chooses an element using the N attribute:

```
child(1,#element,N,2).(1,#element,N,1)
```

Beginning at the location source, the first child (whatever element type it is) with an N attribute having the value 2 is chosen; then that element's first child element having the value 1 for the same attribute is chosen. Non-element node types cannot be chosen because they cannot have an N attribute.

The following example selects the first child of the location source that is an FS element for which the RESP attribute has been left unspecified:

```
child(1,FS,RESP,#IMPLIED)
```

Note that the html keyword is a synonym for a very specific instance of attribute-based addressing such that the following two XPointers are equivalent:

```
html(Sec3.2)
```

```
root().descendant(1,A,NAME,"Sec3.2")
```

3.3.5 The descendant Keyword

The `descendant` keyword selects a node of the specified type anywhere inside the location source, either directly or indirectly nested.

The `descendant` location term looks down through trees of subelements in order to end at the node type requested, *not* down through nested levels of intermediate PIs, comments, or text regions. The search for matching node types occurs in the same order that the start-tags of elements occur in the XML data stream: The first child of the location source is tested first, then (if it is an element) that element's first child, and so on. In formal terms, this is a depth-first traversal.

For example, the following XPointer selects the second `TERM` element with a `LANG` attribute whose value is `DE`, occurring within the element with an `ID` attribute whose value is `A23`:

```
id(a23).descendant(2,TERM,LANG,DE)
```

If an instance number is positive, the search is depth-first and left-to-right. If an instance number is negative, the search is depth-first but right-to-left, in which the right-most, deepest matching element is numbered -1, etc. The order in which elements are examined corresponds to the ordering of the first tag encountered. Thus, the following example chooses the last `NOTE` element in the document, that is, the one with the rightmost end-tag:

```
root().descendant(-1,NOTE)
```

If the last `NOTE` happens to be within another `NOTE`, the containing one is chosen, not the subelement, because it extends to a later point in the document.

3.3.6 The ancestor Keyword

The `ancestor` keyword selects an element from among the direct ancestors of the location source. For positive instance numbers, it counts upwards from the parent of the location source to the root of the containing resource. For negative instance numbers, it counts downwards from the root to the direct parent. Note that `ancestor` can never select the location source itself.

For example, the following XPointer first chooses the innermost element (nearest ancestor) properly containing the location source and having attribute `N` with value `1`, and then the smallest `DIV` element properly containing that ancestor:

```
ancestor(1,#element,N,1).(1,DIV)
```

The node type parameter for `ancestor`, if supplied, must be either `#element` or a particular element type name. If the current location source is an attribute, the element on which that attribute occurs is considered the first ancestor.

3.3.7 The preceding Keyword

The `preceding` keyword selects a node of the specified type from among those which precede the location source. The set of nodes which may be selected is the set of all those in the entire document that occur or begin before the location source. For a positive instance number, it counts left from the location source; for a negative instance number, it counts right from the root element of the containing resource. The first delimiter or tag encountered, starting or ending, counts as an occurrence of that node.

For example, the following XPointer designates the fifth element that occurs or starts before the element that has an `ID` of `a23`:

```
id(a23).preceding(5,#element)
```

Because all ancestors of the location source contain it and potentially other content, ancestors both "precede" and "follow" their descendants. Therefore, the following example selects the root element (probably among other nodes):

```
id(a23).preceding(all)
```

3.3.8 The following Keyword

The `following` keyword selects a node of the specified type from among those which follow the location source. The set of nodes which may be selected is the set of all those in the entire document that occur or end after the location source. For a positive instance number, it counts right from the location source; for a negative instance number, it counts left from the end-tag of the root element of the containing resource. The first delimiter or tag encountered, starting or ending, counts as an occurrence of that node.

For example, the following XPointer designates the second PI that occurs after the element that has an `ID` of `a23`:

```
id(a23).following(2,#pi)
```

Because all ancestors of the location source contain it and potentially other content, ancestors both "precede" and "follow" their descendants. Therefore, the following example selects the root element (probably among other nodes):

```
id(a23).following(all)
```

3.3.9 The psibling Keyword

The `psibling` keyword selects a node of the specified type from among those which precede the location source within the same parent element. The nodes immediately contained by the same parent element are siblings; those siblings which precede the location source are its elder siblings, and those which follow it are its younger siblings.

For a positive instance number, `psibling` counts left from the most recent elder sibling to the eldest sibling. For a negative instance number, it counts right from the eldest sibling. The location term fails if the location source does not have at least as many elder siblings as the absolute value of the instance number.

For example, this XPointer designates the element immediately preceding the element with an `ID` of `a23`, as long as they share the same parent:

```
id(a23).psibling(1,#element)
```

If the location source has at least one elder sibling, then the following location term designates the very eldest sibling:

```
psibling(-1,#element)
```

This location term is synonymous with the following XPointer:

```
ancestor(1,#element).child(1,#element)
```

The value `all` may be used to select the entire range of elder siblings of an element. For example, the following XPointer designates the set of elements preceding the element that has an `ID` of `a23` and are contained by the same parent:

```
id(a23).psibling(all,#element)
```

3.3.10 The fsibling Keyword

The `fsibling` keyword selects a node of the specified type from among those which follow the location source within the same parent element. The nodes immediately contained by the same parent element are siblings; those siblings which precede the location source are its elder siblings, and those which follow it are its younger siblings.

For a positive instance number, `fsibling` counts right from the most recent younger sibling to the youngest sibling. For a negative instance number, it counts left from the youngest sibling. The location term fails if the location source does not have at least as many younger siblings as the absolute value of the instance number.

For example, this XPointer designates the element immediately following the element with an `ID` of `a23`, as long as they share the same parent:

```
id(a23).fsibling(1,#element)
```

If the location source has at least one younger sibling, then the following location term designates the very youngest sibling:

```
fsibling(-1,#element)
```

This location term is synonymous with the following XPointer:

```
ancestor(1,#element).child(-1,#element)
```

The value `all` may be used to select the entire range of younger siblings of an element. For example, the following XPointer designates the set of elements followed the element that has an `ID` of `a23` and are contained by the same parent:

```
id(a23).fsibling(all,#element)
```

3.4 Spanning Location Term

The `span` keyword locates a sub-resource starting at the beginning of the data selected by its first argument and continuing through to the end of the data selected by its second argument. Both arguments are interpreted relative to the location source for the spanning location term itself; the second argument does not use the first argument as its location source.

Spanning term	
[15]	SpanTerm ::= 'span(' XPointer ',' XPointer ')'

Following is an example of a spanning XPointer that selects the first through third children of the element with ID `a23`:

```
id(a23).span(child(1),child(3))
```

3.5 Attribute Location Term

The `attr` keyword takes only an attribute name as a selector and returns the attribute's value.

Attribute-match term	
[16]	AttrTerm ::= 'attr(' Name ')'

3.6 String Location Term

The `string` keyword selects one or more strings or positions between strings in the location source.

String-match term

```
[17] StringTerm ::= 'string(' InstanceOrAll ',' SkipLit (',' Position (',' Length')?)?)'
[18]   Position ::= ('+' | '-')? [1-9] Digit* | 'end'
[19]     Length ::= [1-9] Digit*
```

InstanceOrAll

Identifies the *n*th occurrence of the specified string. For a positive instance number, it counts right from the beginning of the location source. For a negative instance number, it counts left from the end of the location source. For the value `all`, all occurrences of the string are used as candidates in forming the designated resource.

SkipLit

Identifies the candidate string to be found within the location source. A null `SkipLit` string is considered to identify the position immediately preceding each character in the location source. For example, assuming that the element with ID `x37` contains the character string "Thomas", the following XPointer identifies the position before the third character ("o"):

```
id(x37).string(3,"")
```

Position

Identifies a character offset from the start of the candidate string(s) to the beginning of the desired final string match. The position number may not be zero; if omitted, it is assumed to be 1. A positive position number counts right from the beginning of the specified string. A negative position number counts left from the end of the string; for example, position -1 is the position immediately preceding the last character in the match. A position value of `end` selects the position immediately *following* the last character of the match.

Length

Specifies the number of characters to be selected. A length of zero or an omitted length references a precise point preceding the character indicated by Position.

When the location source is a PI or comment, `string` operates on the content of that node. However, the content of PIs and comments is not otherwise considered text content.

For example, the following XPointer selects the position immediately preceding the letter `"P"` (8 from the start of the string) in the third occurrence of the string `"Thomas Pynchon"`:

```
root().string(3,"Thomas Pynchon",8)
```

The following XPointer selects the fifth exclamation mark and the character immediately following it:

```
id(a27).string(5,'!',1,1)
```

For purposes of string matching, the "text of the element" means all the character data in

the element(s) in the current location source and descendant elements. Markup characters are ignored. The pattern matching is exact and character-for-character. No case, space, or combining-character normalization of any kind is to be performed. Thus, there would be no match to "Thomas Pynchon" in the following example. The first seeming match differs in case, and the second by omission of the word-separating space:

```
<example>thomas pynchon,
<auth><first>Thomas</first><family><br/>Pynchon</family></auth>,
Thomas
Pynchon</example>
```

3.7 Locations That Are Not Simply Nodes

Most location terms select a single element as their result: for example, the following XPointer selects one element:

```
id(foo).child(1,SEC)
```

Such cases trivially correspond to nodes in element trees, thus admitting certain implementation simplifications. However, not all locations terms have this limitation:

1. The `string` location term generally returns only part of a node, but if the matched content had markup within it, the result may include portions of multiple elements.
2. The `string` location term, when used with the `all` instance value, returns a list of typically discontiguous portions of string data.
3. The relative location terms may specify the instance argument as `all`, meaning that all candidate nodes are included in the result. The result is thus a vector of possibly non-adjacent nodes, rather than a subtree.
4. A spanning XPointer may include various elements only partially.

Each of these cases is described in more detail below.

3.7.1 Spanning Strings

A `string` location term may return parts of several elements. For example, a `string` that specified the 12 characters beginning at the "c" below would return the entire text content of the `EMPH` element, plus the text region that follows the `EMPH` inside the `P`:

```
<P>Hello, <EMPH>cruel</EMPH> world.</P>
```

3.7.2 The all Keyword

The XPointers shown below specify ordered lists of elements. The elements may or may not be contiguous; in the first case they probably are; in the second, they probably are not:

```
id(sec2.1).child(1,list).child(all,list)
```

```
id(div1).descendant(all,h3)
```

Note that a discontiguous series of elements such as this may be usefully implemented using the same underlying abstract type that would represent the results of a query in certain processing scenarios.

3.7.3 Spanning XPointers

The following spanning XPointer selects everything from the last `P` element in one section through the first `P` in another:

```
span(id(sec2.1).child(-1,P),id(sec2.2).child(1,P))
```

Span locations are not subtrees of XML documents, nor are they mere content data strings. Thus, the result of a spanning selection cannot generally be expressed as a well-formed XML document, nor as a node or list of nodes from an element tree. This is because in general, some elements are neither "in" nor "out of" the span, but in fact are partly in it. For example, the example above includes the end of the element with id `sec2.1`, but not its start. Because of this, implementations that support spans cannot represent them merely as single nodes or as well-formed XML documents; instead they must represent them as pairs of locations or by some other means that can express their greater generality.

Some processing semantics that make sense for nodes or vectors of nodes may not make sense for spans. A browser could easily highlight just the character content of a span, but there may be no appropriate semantics to apply in an outliner or tree-oriented display.

3.8 Link Persistence

It is impossible to guarantee that links to target resources will never break; the resources could be changed in such a way that even the most robust link will break. At worst, the author of a target resource could rewrite it to discuss another subject entirely, making all links irrelevant even if they refer to resources using IDs. However, under typical conditions, some XPointers can be reasonably robust.

The most robust locators are usually those which use only an ID, and this is the preferred locator when available. However, not all elements have IDs, and link creators often do not have enough control over a target resource to have an ID added to it. In such cases the preferred locator is one that points to the nearest containing element that does have an ID, and then walks down the element tree using the `child` location term. This form is relatively robust for two reasons:

- It has a good probability of withstanding editing; for example, no edit outside the element with the ID can harm the reference.
- It will fail obviously rather than quietly if the link does break.

In addition, where relative location terms such as `child` are used, selection by named element type (where the second argument in a relative location term has a `Name` in it) is preferred over selection without specifying a name, for two reasons:

- It is more clear because people typically refer to things by type: "the second section", "the third paragraph", etc.
- It is more robust because it increases the chance of detecting breakage if the original target no longer exists.

4. Conformance

A string conforms to XPointer if it adheres to the syntactic requirements imposed by this specification. Note that this does not require that the string, in association with a URI, actually point to a resource that exists at any given moment.

Application software conforms to XPointer if it interprets XPointer-conforming strings according to all required semantics prescribed by this specification and, for any optional semantics it chooses to support, supports them in the way prescribed. Application software is free to define its own requirements on where XPointer strings will be recognized. For example, an XML application program might choose to recognize XPointers only when they occur in locator attributes of XLink elements.

Appendices

A. Unfinished Work

A.1 Case Sensitivity in Attribute Values

It is possible to specify a link's resource based on the value of an attribute. It is is difficult to decide what the correct behavior is as regards case-sensitivity in matching. Ideally, the declared type of the attribute value should be taken into account, but that presupposes fetching and reading the document's DTD, which may not be appropriate in many XML applications. The current system, while easy to explain, may not prove suitable in the long run.

A.2 XPointers and Abstract Data Types

Formally, the operations of the XPointer mechanism may may be specified as operating on abstract data structures, such as defined in DOM and the HyTime standard ([ISO/IEC 10744]). Every node type in such locators has a corresponding expression in SDQL, and most also have direct equivalents in the HyTime location module.

B. XPointers and TEI Extended Pointers

The XPointer language is based on "extended pointers," a publicly available technology in use by various SGML-based hypermedia applications, defined in the Text Encoding Initiative guidelines [TEI]. This appendix describes how XPointers differ from extended pointers. The main differences facilitate the packaging of locators easily within URIs, and omit some more advanced capabilities:

- Arguments in locator terms must be separated by commas rather than spaces, to facilitate including XPointers within URIs without escapes, and location terms are now separated by periods.
- A spanning XPointer may contain two XPointers separated by the string ",". This combines the capability of the TEI FROM and TO attributes into a single locator syntax for spans.
- The argument-less terms origin and root take an empty argument list, to distinguish them from possible IDs.
- Regular expression matching for GIs and attributes is not included.
- The PATTERN term is replaced by a literal string matching term.
- Options have been added to allow the specification of various non-element node types, including PIs, comments, and portions of unmarked-up text content within elements and CDATA sections.
- In addition, a few terms have been renamed for greater clarity.
- The SPACE, HyQ, and FOREIGN keywords have been omitted.

These changes have been communicated to the TEI, which is considering them for inclusion in a subsequent revision.

Note that the proposed TEI keyword ATTR has been included in XPointer.

C. References

XLINK

Eve Maler and Steve DeRose, editors. XML Linking Language (XLink) V1.0. ArborText, Inso, and Brown University. Burlington, Seekonk, et al.: World Wide Web Consortium, 1998. (See http://www.w3.org/TR/WD-xlink.)

ISO/IEC 10744

ISO (International Organization for Standardization). ISO/IEC 10744-1992 (E). Information technology --Hypermedia/Time-based Structuring Language (HyTime). [Geneva]: International Organization for Standardization, 1992. Extended Facilities Annex. [Geneva]: International Organization for Standardization, 1996. (See http://www.ornl.gov/sgml/wg8/docs/n1920/html/n1920.html).

IETF RFC 1738

IETF (Internet Engineering Task Force). RFC 1738: Uniform Resource Locators. 1991. (See http://www.w3.org/Addressing/rfc1738.txt.)

IETF RFC 1808

IETF (Internet Engineering Task Force). RFC 1808: Relative Uniform Resource Locators. 1995. (See http://www.w3.org/Addressing/rfc1808.txt).

TEI

C. M. Sperberg-McQueen and Lou Burnard, editors. Guidelines for Electronic Text Encoding and Interchange. Association for Computers and the Humanities (ACH), Association for Computational Linguistics (ACL), and Association for Literary and Linguistic Computing (ALLC). Chicago, Oxford: Text Encoding Initiative, 1994.

DOM

Document Object Model Specification. World Wide Web Consortium, 1997. (See http://www.w3.org/TR/WD-DOM.)

CHUM

Steven J. DeRose and David G. Durand. 1995. "The TEI Hypertext Guidelines." In Computing and the Humanities 29(3). Reprinted in Text Encoding Initiative:

Background and Context, ed. Nancy Ide and Jean Véronis, ISBN 0-7923-3704-2.