External entities and alternatives

Karen Lease
SPX FRANCE/Valley Forge TIS, Rueil-Malmaison, France
klease@vftis.com

Abstract:

This talk compares several mechanisms for managing compound documents, including standard external parsed entities, SGML subdocuments, and the recent W3C propositions XInclude and XLink. Based on a detailed discussion of the strengths and weaknesses of each solution, it presents a compromise proposal which leverages existing XML tools.

Introduction

The XInclude mechanism described in a recent W3C Working Draft , is an attempt to invent a new solution to an old problem: breaking complex documents into smaller chunks. Although this problem may concern data-centric documents, the following analysis is primarily concerned with structured documents in the traditional sense. Such documents are generally "authored" as opposed to being created automatically. They frequently have rather complicated schemas (DTDs), and must be maintained over long periods of time. Managing such documents as structured collections of smaller modules facilitates sharing and reuse, and allows revision and configuration management to be handled at the module level, rather than at the document level.

Concepts

The initial XInclude proposition came out of work in the XLink group, which illustrates a certain blurring of the boundaries between these concepts. It is therefore worth examining them more closely. Their semantic model is similar. (See Figure 1 and Figure 2.) An object (**Includer** or **Referencer** below) in **BookEntity** identifies another entity. This pointing object fulfills two roles:

- it locates the point of inclusion or reference in the object where it is contained (Book Entity),
- it identifies the external entity.

Optionally it may identify a particular location in the external entity. What differs is the implied behavior, shown in the lower part of each figure. The relationship between **Book** and the entity identified by the pointer is different in the two cases. Inclusion describes a containment relationship. A book contains its chapters. Reference is a looser relationship between two independent objects. For example, a bibliography describes related publications; it doesn't contain them.

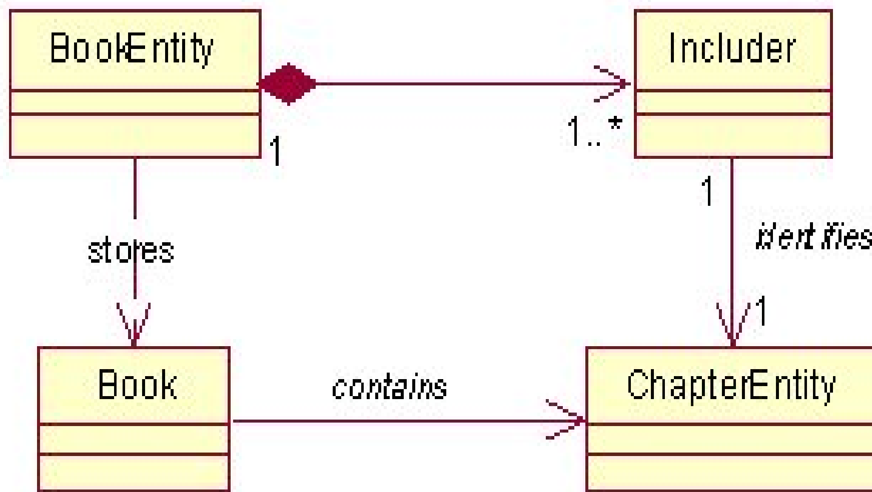

Figure 1. UML representation of inclusion relationships

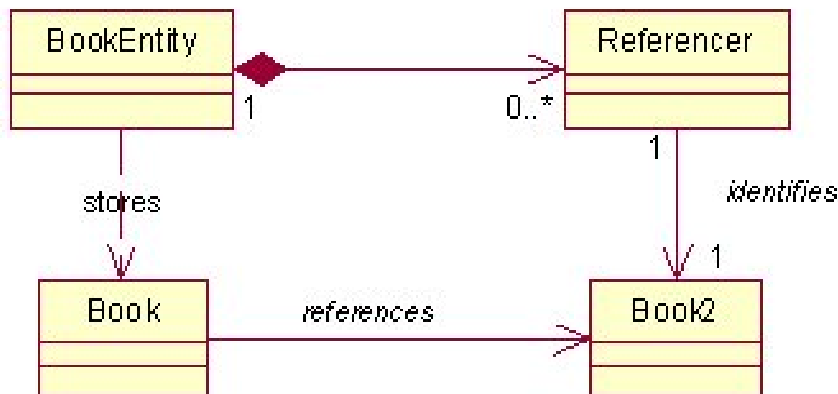

Figure 2. UML representation of reference relationships

Objects in a containment relationship are more inter-dependent than those in a reference relationship:

- the structural integrity of the containing object depends on the presence of its contained objects, but the integrity of a referencing object usually does not depend on the presence of the referenced object. (By presence, we mean the possibility to obtain the identified external content.)
- modifications inside a contained object have an immediate impact on the container; modifications inside a referenced object have little impact on objects which reference them.
- the exact point of reference is crucial in a containment relationship because it identifies the point of inclusion, less so in a reference relationship. (Inverting the order of chapters will confuse readers more than inverting the order of two cross-references.)

For example, suppose a book is stored as a book entity which includes several chapter entities. In an online version of the book, the book entity may be represented as a document containing hypertext links to each of its chapters. Nevertheless, the fundamental relationship is still inclusion and not reference.

Mechanisms

With these concepts in mind, we will compare the XInclude proposal with other mechanisms which can be used to implement modular documents. There are at least three such mechanisms:

- external parsed (or text) entities, as defined by the XML 1.0 Recommendation,
- subdocuments as defined by the SGML `SUBDOC` facility,
- XLinks (with embed behavior), as defined by the W3C XLink Working Draft.

The following table summarizes these four mechanisms. Each aspect is discussed in more detail in the following section.

Syntax	Entity declaration and entity reference (&ref;)	Entity declaration and entity reference or entity attribute value on any element	xinclude element (XInclude namespace)	xlink elements and attributes (XLink namespace)
Reference mechanism	Entity declaration in DTD may contain a public and/or system ID. The system ID is a URI (not a URI reference.)	URI reference (with optional fragment identifier) is specified as an attribute on the include or link element		
Processing model	Entity content is supplied as character data to the XML Processor	By SGML processor (parser) and application	XML Include processor (read as infoset)	By XLink processor and application
Validation context (schema or DTD)	Included material validated in the context of the entity reference	Each subdocument is validated independently	The document is validated before processing the include elements. Optionally the result infoset may also be validated.	Only the XLink elements are validated in the original document; the embedded material is not validated.
Standard	SGML, XML	SGML	XML (WD)	XML (WD)

Table 1. Comparison of modular document mechanisms

Detailed comparison

Syntax

This section summarizes the syntax of the **Includer** object in each model and the restrictions on the entities which it references.

External parsed entities

An external parsed entity is declared in an XML (or SGML) document by an entity declaration without a notation, as shown in the following example. (Entities declared with a notation are **unparsed**, and are not handled directly by an XML processor.)

```
<!ENTITY chap1 SYSTEM "chapter1.xml" >
```

A reference to a parsed entity may occur practically anywhere in a document, between elements or within them, using the syntax `&chap1;`. The entity itself may contain text, complete elements, or a mixture of these. It may not contain any declarations. XML **does** require that entity content be well-formed XML. In other words, one cannot have an element start tag in the document whose end tag is in a referenced entity, or vice versa. This is necessary so that a document may be checked for well-formedness even if the entity references are not replaced.

Subdocument entities

A `SUBDOC` is a special kind of external entity which can be used in an SGML document (but not in an XML document). The declaration looks almost like that of a unparsed entity with the keyword `SUBDOC` replacing `NDATA <notation>`. Although SGML allows `SUBDOC` entity references to occur in mixed element content, it is considered better practice to use them only as the values of `ENTITY`-type attributes on an element.

A subdocument entity has its own `DOCTYPE` declaration, and may be treated as an independent document. Therefore, it may contain only a single subtree of elements.

XInclude

The proposed `xinclude` element is simply an element belonging to a particular namespace. It may be used in any XML DTD (or schema) or any document which declares the namespace containing the element. The `xinclude` element is required to have certain attributes, which are defined by the XInclude Draft proposal. It may also have attributes and content specific to a particular schema. The resource referenced by an `xinclude` element should contain text or well-formed XML.

XLink

The XLink proposal defines a family of attributes in a particular namespace. These attributes may be used with elements in an arbitrary XML schema or document to impose link semantics on these elements. The value `embed` for XLink `show` attribute may be seen as providing a "run-time" include facility. However the referenced resource is not required to be XML. For example, an icon in a document could model an XLink which would "embed" the complete graphic when clicked.

Examples

The following set of examples illustrates the differences between the four methods. In each case, the element `preparation` contains three optional sub-elements, each of which is stored in a separate entity.

Parsed entities

Using external parsed entities the `preparation` element could be coded as follows. The content model of each kind of included element is described by the parameter entities `tools.content`, etc. which are not defined here.

```
<!ELEMENT preparation (tools?, products?, parts?) >
<!ELEMENT tools (%tools.content;) >
<!ELEMENT products (%products.content;) >
<!ELEMENT parts (%parts.content;) >

<!ENTITY include_tools SYSTEM "tools.xml">
<!ENTITY include_products SYSTEM "products.xml">
<!ENTITY include_parts SYSTEM "parts.xml">
.....
<preparation>
&include_tools;
&include_products;
&include_parts;
</preparation>
```

Subdocument entities

Using `SUBDOC` entity references in attribute values would produce the following. Note that the DTD no longer uses `tools.content`, etc.

```
<!ELEMENT preparation (tools?, products?, parts?) >
<!ELEMENT tools EMTPY >
<!ATTLIST tools
    name ENTITY #REQUIRED >
<!- etc ->
....
<!ENTITY include_tools SYSTEM "tools.xml" SUBDOC >
<!ENTITY include_products SYSTEM "products.xml" SUBDOC >
<!ENTITY include_parts SYSTEM "parts.xml" SUBDOC >
.....
<preparation>
<tools name="include_tools"/>
<products name="include_products"/>
<parts name="include_parts"/>
</preparation>
```

XInclude

Using the XInclude proposition as it stands, and assuming that the prefix `xinc` is associated with the XInclude namespace, we would have the following structure. Notice that this conveys much less information than the previous example, as the element name is always the same. If the include element were identified by a marker attribute, as in XLink, rather than by a fixed name, the DTD could distinguish the three kinds of included elements.

```
<!ELEMENT preparation (xinc:include?, xinc:include?, xinc:include?) >
...
<preparation>
<xinc:include href="tools.xml"/>
<xinc:include href="products.xml"/>
<xinc:include href="parts.xml"/>
</preparation>
```

XLink

Finally, using the prefix xlink to designate the XLink namespace, we define the elements tools, products, and parts to be simple links. The use of "marker attributes" rather than fixed element names allows us to clearly distinguish the three kinds of included information.

Note:

This might be considered to be a watered-down version of architectural forms.

```
<!ELEMENT preparation (tools?, products?, parts?) >
<!ELEMENT tools EMPTY>
<!ATTLIST tools
    xlink:type CDATA #FIXED "simple"
    xlink:show CDATA #FIXED "embed"
    xlink:href CDATA #REQUIRED>
<!- ditto for parts and products ->
....
<preparation>
<tools xlink:href="include_tools"/>
<products xlink:href="include_products"/>
<parts xlink:href="include_parts"/>
</preparation>
```

Consequences

The examples show that using SUBDOC, XInclude or XLink elements, a DTD (or an XML-Schema) can define where references to external resources may be placed. Except in the case of xinc:include elements, these techniques also allow a system to define different elements for different kinds of external objects. This is not the case with entity references, as their placement is independent of the document structure. The direct use of external entity references as a mechanism for document modularization therefore requires either disciplined authors or system customization to enforce the use of the right kind of entities in the right places.

When using entity references to represent compound documents, the DTD for the parent document must include the content models for the entities which are to be included. In all of the other mechanisms, there are two variants of the DTD, one which models the elements which indicate the inclusion, and one which models the document when the inclusion elements are replaced by the content of the entities which they identify. Depending on the processing model for the compound documents, this second variant may be implicit.

Identifying the resource

Discussion

The four mechanisms provide different ways of identifying the resource to be included. We will distinguish indirect references (the two entity-based mechanisms) from direct references (the two element-based mechanisms.)

Indirect references

Both general and SUBDOC entity references are meaningless (and invalid) without a corresponding ENTITY declaration in the document's DTD or internal subset. The declaration provides the information which the SGML/XML processor (perhaps with the help of the application) uses to locate the external resource.

In XML, the `ENTITY` declaration may include a public identifier and **must** include a system identifier, which is interpreted as a URI. Note, however, that an XML processor may be signal an error if the URI contains a fragment identifier (see 4.2.2).

References to both parsed and subdocument entities are handled via a processor's entity manager. Most SGML and XML tools provide a way for an application to configure the resolution of public and/or system identifiers associated with entity names in order to handle resource retrieval. For example, when using the SAX interface, an application may register an object implementing the `org.xml.sax.EntityResolver` interface. The `resolveEntity` method of this object is called each time the XML processor finds a reference to an external entity. The method need only return an `InputSource`, which allows an application great freedom in finding the named resource and passing its content to the XML processor.

A single declaration may be referenced multiple times. The entities declared may be modeled by the information set of the document and thus are available to the application. In DOM the Entity and EntityReference interfaces provide access to this information.

Direct references

In contrast, XInclude and XLink elements directly reference the resource with an `href` attribute, whose value is a URI reference. No previous declaration is necessary.

For both XInclude and XLink the resource reference may include a fragment identifier which may be an XPointer expression. Such a reference allows a specific part of the external entity to be retrieved for inclusion (and may in fact designate multiple nodes or even ranges.)

The actual mechanism used to acquire the resource content is outside the domain of these specifications. XInclude and XLink processors could of course provide a mechanism similar to the `EntityResolver` interface to allow applications to intervene in the interpretation of the URI.

Analysis

In the context of document management systems (as opposed to data-oriented applications), the separation of the resource identifier from the point of use is a positive aspect of the entity-based solutions. The use of indirection, as opposed to direct use of URI in attribute values, facilitates the management of version and configuration relationships between entities. Existing authoring tools provide support for management of external entities. For other applications processing entity-based compound documents, the fact that entity declarations are read before the document content means that all potentially-referenced resources are known before processing the document.

On the other hand, the XInclude Working Draft argues that having to declare included resources in advance is unnecessarily restrictive, especially for applications which generate XML on the fly. It may well be that the needs of some dynamic applications are incompatible with the use of "traditional" entity management. But for many such applications, the semantic model is "reference" rather than "inclusion". In this case, the XLink mechanisms would be appropriate.

Referencing sub-resources

While it is clearly necessary to use XPointer expressions in linking constructs, their use in include constructs seems questionable. As discussed in "Concepts", a modular document implies semantic coherence. An included resource is logically **contained** in its including document. In order to guarantee such coherence

a document management system must be able to clearly define what type of content may be included, for example by specifying the allowed schema and root element type. Allowing include constructs to reference bits and pieces of other documents makes it difficult to maintain such constraints. As with indirection, the use of sub-resource references seems to be motivated by a concern with dynamic document generation.

Processing and validation model

From an application point of view, it is useful to have the option of replacing references to included objects with the object content or not, and validating both with and without the included content. The timing of replacement should also be under application control.

Examples

As an example, consider the activity of translating compound documents. Each entity in a compound document is to be translated separately. Therefore, the translation tool does not care what is in included entities and does not need to have access to the actual replacement entity. In fact, if the content of an entity **is** physically present in the documents which include it, then it must usually be protected from unwanted modification.

Many applications can benefit from "lazy evaluation" of inclusions. For example, consider a dynamic diagnostic procedure. Such a document may contain multiple branches, which are dependent on some user action or on a value read from diagnostic equipment. If each branch is stored in a distinct entity, only the one which is actually appropriate in a given situation needs to be retrieved. Any inclusions in this branch are also not processed until needed. If we imagine a complicated diagnostic procedure with many branches, processing the entire scenario at once might be extremely long. In such a case, lazy inclusion becomes highly interesting.

Parsed entity processing

In the case of a validating XML processor, all entity references are assumed to be replaced by the content of the referenced entity. Entity references in the replacement content are recursively expanded. (Note however that the declarations for such entities must have been seen in the original document entity, since declarations are not parsed when reading external entities.) In this case, the replacement content is validated using the schema of the original document submitted to the parser.

A non-validating XML processor **may** choose not to replace an entity reference with its replacement text. In fact, in some cases, it is not even obliged to read external declarations. However, it is assumed to be able to provide information about non-replaced entity references to the application. Both the original XML recommendation and the DOM1 specification mention this possibility. (See section 4.4.3 and "Extended interfaces" in .) The idea is clearly to give applications control over the timing of entity expansion.

Using first generation XML tools, it is difficult to reliably exploit such behavior in an application. Although some parsers provide such functions, SAX1 does not provide a standard interface to instruct a processor not to replace entity references, nor is there an event which allows such references to be signalled to the application. Both of these are however available in the current beta version of SAX2 , so assuming that parser capabilities follow, applications may be able to better exploit this possibility.

Subdocument, XInclude and XLink processing

The other three mechanisms all imply some additional processing in order to replace include references with the referenced content. The general model is shown in Figure 3 below where "Include Processor" is a generic name for the functionality. The result is a new document or infoset which may then be validated against `DTD+` (in which the content models of the include reference elements are replaced by the content models of the entities which they reference.) The mechanisms differ in the level at which this processing is assumed to occur and in the model of interaction between the Include processor and an application.

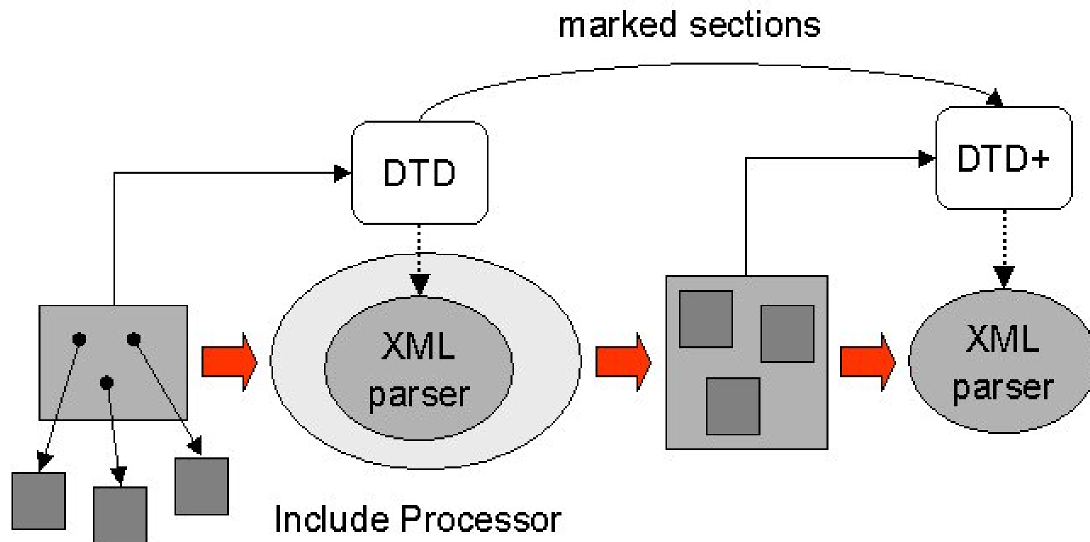

Figure 3. Include processing and validation

Subdocuments

At the level of the SGML processor, the content of an entity referenced as `SUBDOC` does not replace the element bearing the attribute, nor is it even parsed in the context of this element. Eliot Kimber describes the theoretical process as follows:

For parsers, it means providing a mechanism to either parse multiple documents in parallel or to suspend the parsing of the parent document while the subdocument is parsed and then integrating the parsing result of the subdocument with the data resulting from the parsing of the parent document.

The phrase "parsing result" refers in a general sense to the object model (or grove) generated by parsing the subdocument. In practice (James Clark's SP for example), the SGML processor signals the presence of a subdocument reference to an application. The application has the possibility to determine whether or not to "expand" the reference by initiating a nested parsing operation. The application may also choose simply to "remember" the reference for use in a delayed evaluation model.

XInclude

As described in the Working Draft, XInclude processing transforms the infoset of the source document into a result infoset by replacing each `xinclude` element with the content of the referenced resource (recursively.) The implication is that such processing occurs "below" the application level and without its intervention. It is defined as a completely generic processing model. The XInclude processor itself would be responsible for validating the result infoset.

The genericity of this model is what forces the proposal to consider such thorny issues as `ID/IDREF` conflicts and namespace preservation. In a more application-centric model, such issues could be handled in a more flexible way. The stated goal is to avoid the need for each application to rewrite similar code, but is this a domain where "one size fits all"?

The description of the XInclude processing model makes it difficult to imagine how it could be integrated into a SAX-based application. We might be tempted to imagine an XInclude processor as a sophisticated SAX Filter which would acquire the referenced resource, launch a nested parse, perform the XPointer interpretation and then generate SAX events for the next level in the chain. Note that there is no explicit infoset integration here.

The difficulty with this model is that the proposal allows `xinclude` elements to reference locations in the source document ("intra-document references"), and these are required to be resolved with respect to the "original" information set, before any inclusions are performed (see the example in 3.1). Handling of forward references to elements as yet unseen would make it difficult to do this is a single pass.

XLink

Despite the surface resemblance between XInclude and XLink, the processing of an XLink whose `show` attribute value is `embed` is quite distinct from that of an XInclude element. The handling of an XLink is left to the application. The XLink draft states:

...embedding affects only the display of the relevant resources; it does not dictate permanent transformation of the starting resource. For example, if the ending resource is XML, it is not parsed as if it were part of the starting resource.(3.6.1)

This effect could be obtained using an XSL formatting stylesheet. If the XLink has `onRequest` behavior, this can be formatted as a `multi-switch` object in which one `multi-case` corresponds to the non-activated link and the other to the activated state.

Note:

(Digression.) Conceptually, we would like the resource to be retrieved and the corresponding flow objects under the activated multi-case to be generated only when the switch is activated. In practice, this might be difficult, both for XSL template rule interpretation and formatting. We might also ask if the content of the embedded resource should be treated as though it formed part of the original document structure for XSL rule selection.

Conclusion

There are clearly reasons to prefer more structured solutions to parsed entity references when designing modular documents. Improved control over the location and type of included elements could be obtained with any of the alternative solutions. However, I believe that neither XLink nor XInclude (as currently proposed) is adapted to the problem.

XLink

The embed functionality of XLink is aimed at display behavior and not at true inclusion. It may well be that in certain applications, an inclusion will be modeled by an XLink, embedded or not. This may be considered equivalent to using an XSL Link formatting object to represent an included element (for example a chapter in a book.) However, we should not confuse fundamental document structure (inclusion) with its representation

in a particular application (on-line browsing for example.) A document may well model links directly, but the semantics of a link are not the same as those of an inclusion.

Xinclude

XInclude proposes to implement inclusion functionality on the basis of a particular element much as an XLink processor or a HyTime engine exploits particular elements and attributes (or forms). But unlike XLink or HyTime, which are application-level processors, XInclude is presented as a generic sub-application functionality. Assigning a specific processing model to a particular **element** at this level (be it an element with the prefix `xml` as proposed by the Draft) does not seem coherent with the XML architecture.

There is however, a precedent for low-level interpretation of a particular **attribute**: the use of `xmlns:*` attributes to declare namespace domains. The `xmlns` prefix functions as a meta-attribute which is outside of any particular schema, and simply associates a region of the XML document with a namespace URI (and implicitly a schema.) When handled by a namespace-aware XML processor, these attributes affect validation, and in a larger sense, the use that an application will make of the resulting document model.

We might imagine that XInclude functionality could be based on a similar system. The namespace associated with the prefix `xmlinc` would define attributes allowing the specification of a resource to be included. This approach has the additional benefit of allowing a schema to define multiple elements with include behavior.

Why reinvent the wheel?

There is but a small conceptual leap between the notion of an XInclude attribute and an attribute which names a `SUBDOC` entity. Let us suppose that XML were to adopt this concept (with the restriction that a `SUBDOC` entity name is allowed only as the value of an attribute.) In this scenario, the processing could be pushed back into the XML processor level, which already manages entity declarations and entity-valued attributes.

How would it work?

A `SUBDOC`-aware XML processor would be capable of generating an event when the value of an `ENTITY`-type attribute names a `SUBDOC` entity. Using a SAX-style `SubdocumentHandler` interface with the methods `startSubdocument` and `endSubdocument`, an application may interpose its processing model. For example, it might inform the parser to skip the subdocument or to parse it using a new `ContentHandler`.

This approach leverages both existing XML processor functionality and existing APIs. The XML processor can use existing entity management mechanisms to manage resource acquisition. Fundamental processing is managed by the XML processor but an application can easily intervene.

Code reuse can be promoted by creating a library of `Subdocument` filters to cover the common cases of subtree insertion in a DOM or representing an external XML resource as escaped text or CDATA.

Limitations with respect to XInclude

By definition a subdocument entity is a well-formed XML document with a single root element. The XInclude proposal would also require this, since it must parse the addressed resource as XML in order to create an infoset. However, XInclude allows the selection of an arbitrary subset of this document for inclusion. As I mentioned earlier, this kind of functionality seems secondary for true modular document applications. It would, however, be possible to implement it using the filtering scheme described above.

The information concerning the subset could be placed either in the system identifier of the entity (if URI references were allowed in system identifiers), or as part of the attribute value itself (see example below). The particular (and perhaps pathological) case of self-reference in a `SUBDOC` processing model would be handled by recursively parsing the same resource as an InputSource, and does not imply any complicated infoset manipulation.

```
<!ENTITY chap1 SYSTEM "chap1.xml" SUBDOC>
<!ATTLIST incSection ref ENTITY #REQUIRED>
...
<!-include the first section in the entity "chap1.xml" ->
<incSection ref="chap1#xptr(section[1])"/>
```

Going a little further

Finally, for those who really like entities, I will propose a way to use a specific kind of `ENTITY`-valued attribute instead of specific attribute names to handle namespace declarations. This could be done by inventing a `NAMESPACE` entity type. Then a namespace can be declared once and referenced from as many different attributes as necessary, as shown in the following example.

```
<!ENTITY ns1 SYSTEM "http://kdl.com/schema1.dtd" NAMESPACE>
<!ELEMENT kdl:myelem ....>
<?Pub Caret?><!- declare kdl as the prefix for the namespace entity ns1 ->
<!- equivalent to "xmlns:kdl=http://kdl.com/schema1.dtd" ->
<!ATTLIST kdl:myelem
 kdl ENTITY #FIXED "ns1">
```

Bibliography

[DOM1] World Wide Web Consortium, **Document Object Model (DOM) Level 1 Specification**, Recommendation October 1998. **(See http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001)**

[DOM2] World Wide Web Consortium, **Document Object Model (DOM) Level 2 Specification**, CR December 1999. **(See http://www.w3.org/TR/1999/CR-DOM-Level-2-19991210)**

[SAX2] **SAX 2.0beta2 release. (See http://www.megginson.com/SAX/SAX2)**

[WEK1] W. Eliot Kimber, **Re-Usable SGML: Why I Demand SUBDOC**, 1996. **(See http://www.isogen.com/papers/subdoc.html)**

[XINCLUDE] World Wide Web Consortium, **XML Inclusions (XInclude),**Working Draft 22-March-2000. **(See http://www.w3.org/TR/2000/WD-xinclude-20000322)**

[XLINK] World Wide Web Consortium, **XML Linking Language (XLink), Working Draft**, 21-February-2000. **(See http://www.w3.org/TR/2000/WD-xlink-20000221)**

[XML] World Wide Web Consortium, **Extensible Markup Language (XML) 1.0**, REC February 1998. **(See http://www.w3.org/TR/1998/REC-xml-19980210)**

Author

Karen Lease

Senior Programmer
SPX FRANCE/Valley Forge TIS
Postal Address:
Le Consulat
147 av. Paul Doumer
92500 Rueil-Malmaison
France
Telephon: +33 0147 511751
Fax: +33 0147 518714
E-mail: klease@vftis.com

Karen Lease - Karen Lease has been involved in the design and implementation of structured document systems since 1983, on both the programming and the SGML sides. She has extensive experience in writing and using composition software for SGML documents (from Texet to Jade). In recent years, her focus has been on developing tools for preparing hyperlinked technical publications from a variety of source materials including SGML document modules and databases.

Ms. Lease is currently working on Web-based tools for delivering XML and SGML documents. Her main interests include "old" standards such as DSSSL and HyTime as well as their XML-generation equivalents, XSL and XLink.

Ms. Lease previously spoke at SGML Europe 1998 on the subject of "Making an IETM".