Structuring Content with XML

Erik Wilde

ETH Zurich

Abstract

XML as the most successful data representation format makes it easy to start working with structured data because of the simplicity of XML documents and DTDs, and because of the general availabilityof tools. This paper first describes the origin and features of XML as a markup language. In a second part, the question of how to use the features provided by XML for structuring content is addressed. Data modeling for electronic publishing and document engineering is an research field with many open issues, the most important open question being what to use as the modeling language for XML-based applications. While the paper does not provide a solution to the modeling language question, it provides guidelines for how to design schemas once the model has been defined.

1. Introduction

The origins of the *Extensible Markup Language (XML)* lie in document processing, not in data representation in general (even though XML's biggest success today is for data representation for all kinds of data, with the exception of clearly binary data types such as audio or video data). When using XML in its original field, i.e. for representing document content, it is important to look back to the origins, the development, and the core features of the language. A solid understanding of XML, its roots and its design goals make it easier to produce "good XML", even though it would probably be rather hard to define a fixed set of criteria of what "good XML" exactly is (it would probably be much easier to reach consensus on what "bad XML" is).

After this basic look on what for and how XML should be used, Section 3 goes into the details of how these features should be used for representing content structures. Basically, the question is how to define schemas which best represent the content structures (defined by some data model which is outside the scope of this article) that should be captured in an XML document. Again, this is a question of what a "good schema" is, and again, it probably would be easier to agree in what a "bad schema" is.

2. Markup Languages

Structured content in general needs to be represented if it is to be used for machine-based processing. This means that there must be a formal and machine-readable way of representing document structures. In this section, we will look at the most popular way of

representing structured documents, markup languages. Basically, a markup language is a text-based way of mixing structural information (the markup) and content, and the rules for how this is done are defined by every language, or by a framework which enables users to define their own structures.
In Section 2.1, the most prominent languages for defining own structures are presented, which are the *Standard Generalized Markup Language (SGML)* and the *Extensible Markup Language (XML)*. Even though these languages are the most popular ones, there are also some non-markup approaches to representing structured documents, some of which are discussed in Section 2.7.

2.1. A Short History of Markup Languages

Historically, the term Markup comes from a century-long tradition from publishing, where manuscripts were markup up by making notes in the columns or directly in text, and then returned to typesetters for the next revision of the manuscript. This markup consisted of layout information, such as "bold typeface, half-line space afterwards", and because of this repeated process of correcting individual instances of structures (such as headings), manuscript often ended up being typeset inconsistently.
The idea of structural markup (marking up content structures rather than layout information) was first presented by William W. Tunnicliffe at a conference in 1967. The main idea behind this was that marking up the content rather than the layout would make it possible to create consistent layout, and would also allow to reuse the same content with different layout instructions, making it possible to produce differently typeset version of a document from the same source, without the need to modify the source.
Even though Tunnicliffe was the first person to present this idea, the first commercially available and successful implementation of the idea was IBM's *Generic Markup Language (GML)* developed by Charles Goldfarb, Edward Mosher and Raymond Lorie in 1969. GML extended Tunnicliffe's general ideas with a well-defined nesting scheme of document structures, and the idea of document types, which define the allowed structures for a class of documents (more about document classes in Section 3.3). Interestingly, what later became known and successful as GML was called *Text Description Language* first (as an internal name), and this still captures the essence of what markup languages are about: Describing a text's contents on the structural level, making it possible to have different applications using these descriptions for application-specific purposes.
Brian Reid's Scribe system developed as part of his doctoral thesis in the late 70s also made important contributions to the field of markup languages, in particular the idea of styles which were layout instructions being kept completely separated from document content, so that styles could be easily exchanged.
Systems such as the Unix formatter *troff* and the T_EX [34] typesetting program became available in the late 70s and early 80s, and they also underwent the transition from pure typesetting systems to markup-style systems, using logical structures instead of layout instructions. For troff, this was implemented by several popular macro packages, for TEX there is only one noticeable macro package, which is $L^A T_EX$ [35].

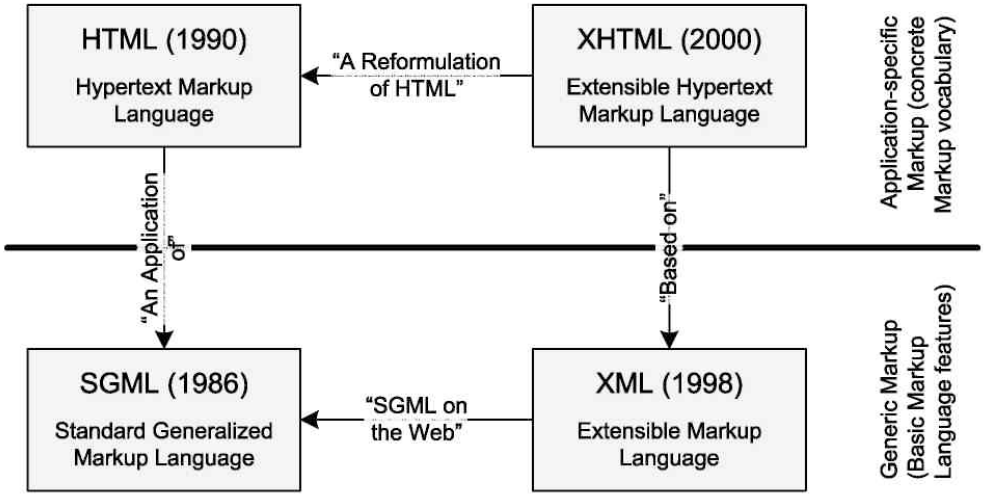

Figure 1: Markup Language Concepts

Continuing the work on GML and joining forces with Tunicliffe and Reid, the GML team moved on to develop the *Standard Generalized Markup Language (SGML)* [22], which became the most important markup language and an important foundation for many document processing environments, because it was a standard rather than a product. This meant that using SGML, customers would not be bound to a specific software or vendor, and (at least theoretically) it would be possible for everybody to implement an SGML system by simply implementing the ISO standard. However, as it turned out, SGML was overloaded with obscure and hard-to-implement features, which were only required by few users. As a consequence, many implementations of SGML implemented only parts of the standard, which made many SGML implementations non-interoperable.

However, even though SGML has had some problems with complexity and interoperability, it was the single most successful format for structured documents, and big organizations started recognizing the importance to store their documents in a well-defined and reusable way. For example, the U.S. Army required contractors to submit their documentation in SGML, which made it reusable and independent from any specific software or vendor.

Of course, when marking up document content, the question is what structural concepts do exist, and how can they be combined? For example, if there are sections which consist of headings and paragraphs or lists, would it also be allowed to use paragraphs within a list? These questions are addressed with SGML's concept of a *Document Type Definition (DTD)*, a definition of a class of documents which adhere to a well-defined set of rules. Using this concept, first a DTD is designed, and then instances (i.e., actual documents) of this DTD can be created, which must adhere to the rules defined in the DTD. More on document classes and DTDs can be found in Section 3.3. Seen this way, SGML is not a

markup language in itself, but a mechanism to define vocabularies for marked up documents. This is shown in Figure 1, which also shows some of the more recent relatives of SGML.

In the beginning of the 90s, markup languages became the driving force of the globalization of the Internet with the *Hypertext Markup Language (HTML)* [47], the content language of the *World Wide Web (WWW)*. In its first versions, HTML was more inspired by SGML than being based on it formally, but this was changed later and HTML became a document type of SGML, which in SGML terminology is called an "SGML application". Because of its ad-hoc creation and rapid success, HTML still incorporates some design oddities from its early years, for example the following points:

- *Containers:* In most cases, markup languages will define containers for sub-structures, such as a container for sections, which then holds the section's heading as well as its contents. HTML uses a more linear structure of contents, allowing different levels of headings (using structures named `h1` through `h6`), but the sections themselves are not contained in any markup, they are only implicitly defined. This makes it easier to type in HTML, but makes it harder to edit and process it.
- *Emphases:* Text in HTML can be emphasized in two ways, either by using logical markup (`em` for emphasized text and `strong` for strongly emphasized text), or by using layout-oriented markup (`i` for an italic typeface and `b` for a bold typeface). Newer HTML version declare the layout-oriented markup as deprecated (and encourage users to use CSS for specifying layout information), but this parallelism of the logical as well as layout-oriented markup still confuses users of HTML and allows documents to use layout-oriented markup instead of logical structures.

Even though HTML may not be perfect from the markup design point of view, its simplicity makes it easy to create. HTML's balance of simplicity and expressive power obviously was a one of the major reasons of the Web's success, and even though many HTML creators today do not know anything about markup languages and their concepts, they use an SGML-based markup language, as shown in Figure 1.

Even though HTML is well-suited for marking up documents for Web publishing, it is much less suited for marking up documents for general-purpose document processing. HTML basically is a rather simple page description language for Web pages, and as such has many limitations, such as missing support for paginated media, better layout control in general, or more detailed text structures apart from the simple model of paragraphs and simple lists. In the 90s, the Web grew at an astonishing rate, and there was a clear demand from content providers to have a better way of representing their content than just HTML. In particular, the trend towards mobile computing made it clear that contents should be kept in a device-independent way, and HTML would only be one way of publishing content (other possible channels being formats for mobile devices, or formats for print-quality publication).

The main body for Web standards development, the *World Wide Web Consortium (W3C)* (more about standardization bodies in Section 2.4) therefore started an activity called "SGML on the Web", with the goal of defining a language which would provide the

flexibility of SGML (in particular, the ability for users to define their own markup vocabularies), but avoid the complexity of the full SGML standard. The outcome of this activity was the *Extensible Markup Language (XML)* [10, 9], which functionally is a subset of SGML [12].

XML supports the main concept of SGML, the *Document Type Definition (DTD)* which enables users to define their own classes of documents (as described in Section 3.3). However, XML leaves out many of the features that were thought of as being of only limited use. In fact, even though XML is radically simplistic in comparison to SGML, from today's point of view (7 years after XML was first published) probably some more features would probably be left out, and a cleaner processing model would be defined.

However, even though XML may not have been the perfect markup language (in fact, it also is a generic markup language because it is a mechanism for defining vocabularies and not a markup language in itself), it has had a lot of success. Some of the reasons for this are political (it did not come from one of the major vendors and as such is free of any operating system or programming language legacy), and others are technical (the Internet needed a format for exchanging structured information). XML's success has been astonishing, and availability of a universally accepted format for exchanging structured information often vastly outweighs any minor concerns users might have with some details of the language.

It is interesting to note that even though XML was designed and thought of as a document structuring language (in the same way as SGML was intended for documents), the majority of XML users today are using XML for exchanging data rather than documents. Data in this case is more general than documents, in the sense that documents are a certain type of data characterized by properties such as instance variability and semi-structured models, while data can be anything, but in practice often are just well-defined structures such as tabular material for typical business-to-business applications exchanging business data such as orders and invoices.

The last building block shown in Figure 1 is the *Extensible Hypertext Markup Language (XHTML)* [45], which has been defined in 2000. Because HTML is based on SGML, and XML is supposed to be the foundation for all machine-readable data in the future, the W3C decided to create a new version of HTML. This new version is XHTML, and it is important to understand that XHTML does not add new features to HTML, it simply defines XHTML as being based on XML, so that XHTML documents can be processed using normal XML tools. This way, XHTML fits nicely into the overall picture of structured documents being XML documents, rather than using the older SGML syntax of HTML, which is much harder to process.

2.2. Using XML

After this short history of markup languages, we will now look at how to use markup languages. This discussion is mainly based on XML (with some remarks about SGML to better understand how HTML works), but it is not meant to be a comprehensive description of the XML specification.

2.2.1. Basic Structures

In most markup languages (and in XML and SGML), content structures are modeled as a tree. This means that overlapping structures are not possible, everything must be properly nested, and if structures should be defined which cannot be represented as a tree structure, then this is simply impossible with the tree-based approach (see Section 2.3 for a discussion about alternatives). Figure 2 shows such an example, using (X)HTML[1] elements for paragraphs and emphasized text. If the text segment spanning the end of the first paragraph and the start of the second paragraph should be emphasized, then the most natural way would be to do it like shown in the second alternative. This, however, is illegal, because the first paragraph now ends before the emphasis ends, resulting in an improper nesting. The third alternative shows a way how this could be done in HTML, but it should be noted that this is not really the same as the second alternative, because it uses two emphases rather than the originally intended single emphasis.

(1) Paragraphs without Emphasis:

```
<pseudohtml>
  <p>This is the first paragraph.</p>
  <p>This is the second paragraph.</p>
</pseudohtml>
```

(2) Paragraphs with illegal Emphasis (not proper Markup):

```
<pseudohtml>
  <p>This is the <em>first paragraph.</p>
  <p>This is</em> the second paragraph.</p>
</pseudohtml>
```

(3) Paragraphs with Tree-like Emphasis:

```
<pseudohtml>
  <p>This is the <em>first paragraph.</em></p>
  <p><em>This is</em> the second paragraph.</p>
</pseudohtml>
```

Figure 2: Markup is limited to Tree Structures

[1] Since HTML and XHTML use the same elements, there is no semantic difference between these languages, and the term HTML will be used from here on. However, the following examples all use XML syntax (except when specifically marked as being non-XML) and thus are HTML- as well as XHTML-compliant.

This limitation to tree structures is one of the most fundamental things to get used to when working with markup languages. Trees can be as complicated and deeply nested as required (and allowed by the application), but they need to be properly nested trees in order to be represented as markup.

Figure 2 shows the most important structural construct of XML, which is the element. Elements can be thought of as containers, and each node in the structural tree of a documents is represented by a start tag (the element name enclosed in angle brackets), followed by the node's contents, and the end tag (angle brackets enclosing a slash and the element name). This definition can be applied recursively, until there are no nodes left containing other nodes (they may contain text only, or they may be empty), and this way a arbitrarily complex and deeply nested tree can be represented as a sequence of tags and content. Which is exactly what markup languages are for, representing structures in purely textual form (see Section 2.7 for a discussion of non-textual alternatives).

Using this simplified model of XML, an XML document simply is a tree of elements, which may contains other elements and/or textual content. Figure 3 shows an actual XML document, which shows a number of additional concepts which are important for markup languages and XML in particular.

```
<?xml version="1.0" encoding="UTF-8"?>
<document created="20050226" modified="20050315">
     <author>Erik Wilde</author>
     <title>Markup Languages</title>
     <p>Non-linear content (as discussed in the previous section)
          and structured content in general needs to be
          represented...</p>
     <p>In Section <xref id="using-markup"/>, it is explained
          how...</p>
     <section id="using-markup">
          <title>Using Markup Languages</title>
          <p>After this short history of markup languages, we will
          now look at how to use markup languages...</p>
     </section>
</document>
```

Figure 3: XML Elements and Attributes

- *XML Declaration:* In order to make a document recognizable as XML, it should contain an "XML declaration", which clearly marks the document as being an XML document. The declaration indicates the version of XML, and also the character encoding being used, which is important because XML supports the whole Unicode character repertoire. How characters (especially those outside the ASCII character range) are encoded is important for correctly interpreting the document, and therefore should be specified in the XML declaration as well.

- *Application-specific Structures:* The element names (such as `document` and `author`) are not part of HTML (or any other standard vocabulary), and in fact have been made up for this example. This is the most interesting aspect for most XML users: It is possible to define and use own vocabularies for structuring content, thus making it possible to adapt XML to a wide variety of application areas. The actual mechanism of how these vocabularies are defined are described in Section 3.4, and the concrete example for the example XML document is shown in Figure 9.
- *Attributes:* While the elements clearly form a tree structure, some of them have additional information associated with them, which are called attributes. Attributes are a method of specifying additional information for an element, which is not considered content of the element (this definition is very vague and often the source of confusion, but there are no well-defined rules when to model information as an element or as an attribute). While attributes are additional nodes in the tree (they are associated with the element on which they appear), they cannot contain structured information (from the XML point of view), which means that attributes may not contain elements.
- *Empty Elements:* The `xref` element in the second paragraph is an empty element (it is meant to represent a cross reference to a section, which probably be formatted as a section number of a hyperlink, depending on the target format). There are two interesting observations: Firstly, this means that empty elements can be useful, because their mere presence is sufficient to convey the necessary information. Secondly, it shows that there is a shorthand notation for empty elements, which collapses both the start and the end tag into a single empty tag, which is indicated by a slash before the closing bracket.
- *Minimal Semantics:* The `id` attribute on the `xref` element and the `id` attribute on the `subsection` element are meant to represent a cross reference and an identifier (which is referenced in the cross reference). XML has features to declare identifiers and references to them, which ensure that identifiers occur at most once, and that references to identifiers refer existing identifiers only. This way, XML ensures referential integrity without any need for the application to check for these constraints.

These basic concepts of XML cover the essential parts of XML. It is this combination of a very simple model (trees using elements with content and attributes, nested as deeply as possible) and a very flexible architecture (elements and attributes and their usage are user-defined), which in combination are one of the main factor for XML's success.

When thinking of trees and XML as a representation of these trees, there is an apparent similarity to the ideas of conceptual and logical models as known in the relational database world. While conceptual models are used to model the universe of discourse in an abstract way, using *Entity-Relationship modeling* or derived mechanisms, logical models are implementations of this abstract model on the database level, dealing with implementation questions such as tables and columns.

XML can be regarded as being on the logical model level, because using XML requires implementation details such as to decide whether a particular property should be modeled

as an element or an attribute. Unfortunately, so far no generally accepted conceptual modeling method has emerged, even though first steps are being taken [40]. Until then, conceptual models of XML will need to use informal or application-specific methods, and then translate these into XML.

2.2.2. Additional Structures

The elements and attributes introduced in the previous section are the most important structural components of XML. However, XML also supports other structural components. To make this discussion independent from implementation details of XML, it is more appropriate to talk about an abstraction of XML, which is the *XML Information Set (Infoset)* [15]. Basically, the Infoset is an abstract definition of the tree structure represented by an XML document, and leaves out certain details (for example, the order of attributes within a start tag, or the white space characters within tags). Most recent XML technologies in fact are based on the Infoset (or derived models) rather than XML, because it is easier to work on this abstraction rather than having to deal with every single XML way of representing structures [58] (additional information about the Infoset can be found in Section 3.1).

```
<?xml version="1.0" encoding="UTF-8"?>
<?html layout="compressed"?>
<?pdf layout="normal"?>
<!-- this is a sample comment -->
<document created="20050226" modified="20050315">
  <!-- comments may also appear in element content <nomarkup/> -->
  <author>Erik Wilde</author>
  <title>Markup Languages</title>
  <p>In Section <xref id="using-markup"/>, it is explained
  how...</p>
  <section id="using-markup">
    <title>Using Markup Languages</title>
    <math:math xmlns:math="http://example.com/math">
      <math:equation>
            <math:left>x</math:left>
            <math:right><math:sqrt>1764</math:sqrt></math:right>
      </math:equation>
    </math:math>
  </section>
</document>
```

Figure 4: Examples of all XPath Node Types

For many XML developers, the model defined by the *XML Path Language (XPath)* [11] is the model they are working with, because this is the model they use when working with the XML transformation language *XML Transformations (XSLT)* [13]. In fact, the XPath model is derived from the Infoset also, and it models XML documents as consisting of seven different node types (all of which are represented in XML in Figure 4):

- *Document:* Each XML document has one document node, which is the root of the tree and thus the (virtual) container for all content of the document. The document node has no physical correspondence in XML markup.
- *Element:* Each XML document has exactly one top-level element, which is called the *document element*. Elements are the most important nodes and are the structural components which make XML so powerful and flexible, because they can be nested. Apart from containing other elements, elements may also have attributes, or they may contain text, comments, or processing instructions.
- *Attribute:* Attributes appear on elements, but they are generally not considered to be content of the element. An attribute has a name and a value, and the value is a character sequence, it may not contain markup such as elements, comments, or processing instructions.
- *Text:* Text nodes appear within elements, and they may be mixed with other markup. If text is mixed with elements, then the containing element is said to have "mixed content", because the textual content is mixed with elements. A typical example is the `p` element from Figure 4, which contains a sequence of text, the `xref` element (which is empty), and more text. Consequently, the `p` element has mixed content.
- *Comment:* Comments may be used within XML documents and they start with `<!--` and end with `-->`. "Markup" within comments is allowed, but not recognized as markup (which makes comments a handy tool to comment out parts of an XML document). Even though comments are clearly part of the XML document within which they appear, they are generally considered irrelevant for future processing of the document.
- *Processing Instruction:* Processing instructions are used for processing-specific information, which is not considered to be actual content of the document, but nevertheless should be available to applications processing the documents. Processing instructions have the same syntax as the XML declaration (which technically speaking is not considered a processing instruction), they start with a `<?`, followed by a so-called *target* and the processing instruction's contents, and a `?>`. Very often, the processing instructions will take on the syntax of XML attributes (as the `layout` parameter in the example), but technically speaking processing instructions simply have content which is a sequence of characters. The target is used by later processing stages to decide which processing instructions should be interpreted, and often specifically identifies certain steps of the document processing pipeline.
- *Namespace:* Namespaces are discussed in detail in Section 2.2.3. They are a mechanism for associating names (of element and attributes) with a *namespace name*, so that these names can be identified as belonging to this particular

namespace. In the example, the elements prefixed with `math:` belong to the namespace `http://example.com/math`, which is declared by using the namespace declaration `xmlns:math="..."`.

This concludes the list of structural components which are important in an XML document. This still gives a slightly simplified view of XML, because it does not address the issues which are not visible through the Infoset/XPath view of an XML document which has been used here. However, since the detailed knowledge of XML on the syntax level is beyond the scope of this section (and this book), these issues will not be addressed.

2.2.3. XML Namespaces

The basic model of XML and the way the
core XML specification has been designed assume that the elements and attributes used for structuring content come from one vocabulary. Practice has shown, however, that in many real-life cases, different vocabularies are mixed in one document. The most popular example are Web Services, which are designed in a way that the basic structure of a message uses SOAP as the Web Service standard for encoding messages, whereas the actual payload of the message is encoded using an application-specific vocabulary. Thus, every Web Service message combines different vocabulary, the SOAP vocabulary for the overall structure, and another vocabulary for the application-specific aspects.

Another example is HTML, which has been designed as a general language for describing structures of Web pages. HTML has virtually no support for the advanced layout capabilities required for mathematical formulae, but the *Mathematical Markup Language (MathML)*, described briefly in Section 2.6, has been designed for marking up mathematical formulae. Consequently, MathML markup could be embedded into HTML, and a browser capable of processing HTML as well as MathML could render Web pages using complex mathematical formulae. Figure 4 shows a scenario similar to this, by embedding a hypothetical `math` vocabulary into the overall document structure.

XML Namespaces have been designed to support this kind of application, where different vocabularies have been mixed within one document, and must be identified as different vocabularies for processing. As shown in Figure 4, this is done in two steps:

1. *Declaring a Namespace*: A namespace is declared by using an attribute, the attribute name is either `xmlns` (declaring the default namespace), or `xmlns:` followed by a name (the so-called *namespace prefix*). This declaration is in scope for the element on which the declaration has been used, and the contents of this element (in particular, elements within this element). The important aspect of the namespace declaration is the fact that it declares that names from the declared namespaces could be used now. The prefix is only of local significance, it is merely a connection between a declaration, and the usage of a name from the declared namespace.
2. *Using a Declared Namespace:* Within the scope of the namespace declaration, names from the namespace can be used by prefixing them with the declaration's prefix. If the namespace has been declared as the default namespace, names need not need to be prefix (in fact, there is no prefix for the default namespace), and element names with no prefix are interpreted as coming from the default

namespace (as a complication, the default namespace does not apply to attributes). As shown in the example, if a prefix is used, all occurrences of names from the namespace (in particular, in start and end tags) must be prefixed.

Even though this way of declaring and using namespaces is quite simple, it allows a wide variety of different styles to declare and use namespaces, which can be confusing. Figure 5 shows the relevant fragment of the original example in two different, but equivalent ways.

(1) Namespace Declaration with a Prefix:

```
<section id="using-markup">
    <title>Using Markup Languages</title>
    <math:math xmlns:math="http://example.com/math">
         <math:equation>
            <math:left>x</math:left>
              <math:right><math:sqrt>1764</math:sqrt></math:right
            >
           </math:equation>
      </math:math>
  </section>
```

(2) Default Namespace Declaration:

```
<section id="using-markup">
      <title>Using Markup Languages</title>
      <math xmlns="http://example.com/math">
           <equation>
             <left>x</left>
             <right><sqrt>1764</sqrt></right>
           </equation>
      </math>
</section>
```

Figure 5: Equivalent Ways of Using Namespace Declarations

In the first variant (from the original example), the namespace is declared with a prefix, and this prefix is then used for all elements, marking them explicitly as being from the declared namespace. In the second variant, the namespace is declared as the default namespace, which means that within the scope of the declaration, all unprefixed elements will be interpreted as being from this default namespace. Consequently, the elements now do not need a prefix anymore. In comparison, the first variant requires more markup, but is more explicit when looking at the document, while the second variant minimizes the required markup, but makes it harder to recognize that a particular element (such as the `right` element) actually is from another namespace than the document element.

One question that has not been discussed so far is the question of where namespace names come from. Namespaces names must be URIs, but apart from that, no restrictions apply. There is no registry of namespace names, and vocabulary designers can simply invent namespace names for their vocabularies, without the need to check with any authority. However, it has become customary to use namespaces names using the own domain name, which in many cases makes it comparatively easy to see where a namespace has been defined. For example, the W3C has defined the namespace name `http://www.w3.org/1999/xhtml` for XHTML, which means that every HTML page that claims to be XHTML must use the element names from this namespace.
Finally, there is the question of whether the namespace name (which is an URI) must refer an actual resource. The namespaces recommendation says that this is not the case, so it is legal to define and use a namespace name which, when used as a URI for retrieving a resource, does not succeed. The only way for an application to implement support for a namespace thus is to have built-in knowledge about the namespace name and the associated vocabulary. However, it is good practice to use a namespace name which actually identifies a resource. This resource then either is an HTML page describing the namespace and pointing to relevant documents, or it may be a machine-readable resource, but so far there is no established format for what format to use. The Web architecture [31] published by the W3C recommends that namespace names should point to resources, but leaves open the question of a particular format for these resources.

2.2.4. XML Application Scenario

The basic technologies for using markup today are XML as described in Section 2.2.1 and XML Namespaces as described in Section 2.2.3. The question now is how these static descriptive structures fit into the process of a generic XML processing pipeline, or more specifically into the workflow within the electronic publishing process. Generic XML processing in its typical stages is briefly discussed here. The first question is where documents are stored. Looking at the possible ways how documents may be fed into a processing pipeline, the following three main approaches are possible:

- *File System Storage:* Documents can be stored in a normal file system, either as monolithic documents, or in some modularized way, as described in Section 3. In both cases, the document is retrieved from the file system and then processed.
- *Database Storage:* File systems often lack the flexibility, descriptive power, and scalability required for storing content. Thus, in many more advanced scenarios database systems are being used. While database systems are designed for any type of data, a *Content Management Systems (CMS)* is a specialized database system, that has been built for storing, managing, and retrieving documents or document fragments.
- *Dynamic Content:* While the former two approaches are basically static data (as long as the file system or database is not updated), electronic publishing often also includes dynamic content, typical examples being prices from stock exchanges, or newstickers, which by their very nature do not have any static state or content. In these cases, the content is extracted from a source which always returns the most recent content.

Of course, these approaches can also be combined, for example when data coming from dynamic data source is combined with static content from a file, to form a document with a static frame and dynamic content (this approach is very often used when producing Web content, where the overall frame of the document is static, while the contents are generated from a dynamic source such as a database query).

After the document to be processed has been assembled one way or the other, it is assumed that it uses some kind of markup language, for example XML. It is first being processed by a *parser* (the XML specification calls this an *XML processor*, but the term *parser* is in much wider use). A parser takes as input a markup document, and interprets and thus checks the markup. If there are markup errors, the parser will reject the document. Optionally, the parser may also check the document against a document class (as discussed in Section 3.3), and will reject the document if it does not conform to the constraints of the document class. The result of a successful parsing process is a structured representation of the document.

This structured representation can either be an in-memory representation of the document, or a stream of events which represent the document. Figure 6 shows the schematic diagram of how an XML processor is working, and in particular the interfaces through which applications have access to the structured representation of the parsed document.

Two typical interface types of a parser is the event-based interface type provided by the *Simple API for XML (SAX)* interface, and the tree-based interface type provided by the *Document Object Model (DOM)* [38]. While SAX is a lightweight but limited interface to documents, which basically reports all recognized markup structures to the application in the form of events, more complex application may want to have random access to the document's content, in which case a tree based interface which allows the application to access the in-memory tree representing the document.

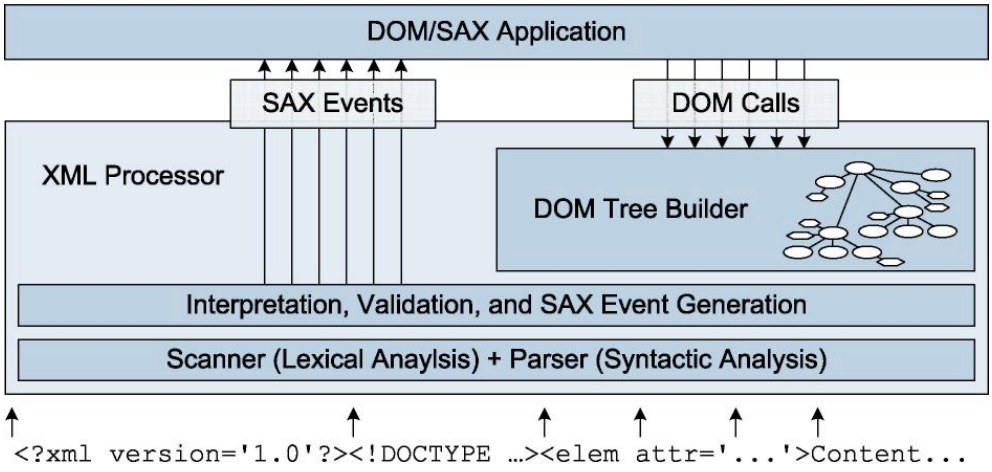

Figure 6: XML Processor with DOM and SAX Interfaces

The choice between the two types of interfaces depends on the application as well as the documents. Complex application requirements (random access, many tree-oriented operations such as visiting neighbor nodes) may make a in-memory tree more convenient. However, the memory requirements of such an interface type are substantial (because the whole document must be represented as an in-memory data structure[2]), and if the documents are very large, it may become impractical to use a tree-based interface. Thus, the choice of an appropriate parser and interface for processing XML documents is an important decision, but can not be made without knowledge of the application scenario in terms of document access and document size.

2.3. Shortcomings and Alternatives

While the general idea behind markup languages as described in Sections 2.1 and 2.2 is a rather generic concept and useful for many applications scenarios, it still is a model with limitations. For example, Ted Nelson, the inventor of the term "hypertext" and very influential in the hypermedia community, has claimed that the concept of embedded markup in itself is a bad idea and should not be considered a good solution for hypermedia systems [44]. The reason for this is that a structure of a text already is an interpretation of a text, and since there can be different interpretations (and therefore structures), they should be considered as something separate rather than something embedded.

In general, the concepts of markup languages such as SGML and XML have made working with structured content much easier, but have also imposed a certain world-view on their users, which might not be appropriate for all application areas. It is important to keep in mind that SGML or XML — as versatile and flexible as they are — are not the ideal solution for every problem. In particular, the following limitations are obvious:

- *Restrictions of Document Structures:* The tree-based model of SGML and XML is a relatively versatile model that can be used to represent many different application data structures, but it is only a subset of possible graph structures and as such may be considered too limited in some cases. The cross-references in the document tree (by using `ID/IDREF` attributes) may be used to augment the tree with additional structure, but also have only limited expressiveness and lack some basic features (in DTDs, `ID/IDREF` are always global concepts which cannot be restricted to element types).

  Figure 7 shows an example where the same content (taken from Figure 2) should be structured with two different "views" of it (two different interpretations of the same content), and there is no obvious solution how to handle this with basic SGML or XML.
- *Restrictions of Document Class Definitions:* While XML and SGML itself impose some limitations on what can be represented easily in a document (i.e., in a tree), the DTDs impose even more restrictions on the way document classes (i.e., classes

[2] There are parsers that have options which allow users to choose between different memory models, trading space against access time, but even then there may be documents of a size where even the smallest memory model will require too much space.

of trees) can be defined. The features and limitations of DTDs (as well as possible alternatives) are discussed in detail in Sections 3.4 and 3.4.1, the most important limitation being that XML and SGML are based on the concept of context-free grammars, which means that any context-specific concept is hard to capture.

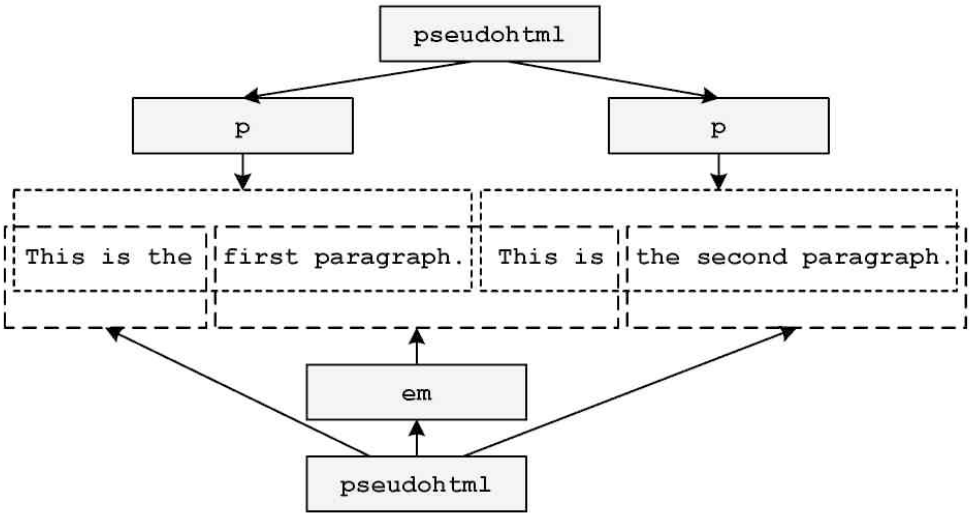

Figure 7: Multiple Content Structures

Keeping these restrictions in mind, it becomes clear that markup languages such as SGML and XML may not be the ideal solution for all problems. The example shown in Figure 7 is one such case, where the application requirement is two support two alternative interpretations (i.e., structures) of the same content. XML does not allow this at all. However, there is the CONCUR feature of SGML, which unfortunately is not supported by all SGML implementations, so even though this would be a viable solution, in practice it also could be compromised by questions of portability and application support.

Questions such as this come up in the document-oriented XML community on a regular basis, and solutions such as *Just-In-Time-Trees (JITT)* [17] or the *Layered Markup and Annotation Language (LMNL)* [50] have been proposed. However, these solutions have never been adopted by any standards organization or significant user base, and thus are an indication that the problem has been identified and solved in specific application areas, but so far has not been big enough to be addressed by a larger community or body.

Section 3 discusses the general problem of how to structure content using markup languages in greater detail, and in particular described the *XML Linking Language (XLink)* (Section 3.7.3). Using this language, it would be possible to solve the problem shown in Figure 7 with a standards-based approach, but the solution would not be as elegant as the proprietary JITT or LMNL approaches mentioned in the previous paragraph.

2.4. Definition and Usage of Markup Languages

When discussing technologies in the markup language area, it may be important to look at the source of these technologies. The reason for this is that depending of the source of the technology, its lifetime and support by various vendors probably will be different. However, even if a technology has been standardized by a recognized body, it may not be accepted by the marketplace and may never be adopted by a significant number of vendors. Thus, it is not sufficient to look at the source of a technology, it is also important to look at a technology's adoption in the marketplace. However, the source of a technology remains an important aspect of the overall judgement of a technology's likely adoption, stability, and longevity:

- *Standards Bodies:* Standards bodies usually are more or less open to participation from interested parties, even though the barriers to entry are very different. Typically, a standards body will try to achieve a consensus among the parties which are involved in the standardization, and many standards suffer from this by having many optional parts which have been accepted as a tradeoff between different parties. The advantage of a standard is that is it under the control of the standards body, which has public and often reasonably robust rules for maintaining the standard, so that it can be expected to have a long lifespan. However, there is nothing that guarantees a standard's success, and there are many examples of failed standards which never got any traction.
- *Informal Bodies:* While standards bodies are the traditional and widely accepted source of defining consensus about technologies, the slow action and consensus-oriented nature of the standardization process can be frustrating for practitioners trying to get something done. The collaboration technologies available through the Internet (mailing lists, email archives, discussion forums) have made it possible for people with limited funds and a focused interest to interact and achieve consensus about technologies on a less formal, but often more efficient basis. Technologies defined by informal bodies sometimes become very successful because they serve an obvious need from the user community that has not been addressed by standards. Technologies from informal bodies can become stable and widely accepted (SAX as discussed in Section 2.2.4 is one such example), but they often suffer from instability or insufficient reviews from potential users.
- *Companies:* Rather than going through the time-consuming standardization process or using potentially instable informally defined technologies, companies often decide to take their own technologies and brand them as "standards". In some cases these industry standards may become very successful and widely used, but it is important to keep in mind that they are still under the exclusive control of a company, which often has numerous patents covering *Intellectual Property Rights (IPR)* and allowed terms of usage. Also, the company may decide to develop the technology in a direction different than that hoped for by some users, and unless they can build relevant market pressure, they will not have any possibility to influence the technology's development. Thus, before building information architectures (and especially those which should have a very long lifespan) on top of industry standards, this decision should be viewed with a worst-

case scenario in mind, such as the company leaving the market or being taken over by the competition which favors another (probably its own) technology.

While the different sources of specifications and technologies may look confusing, they often mirror interests, conflicts, and how to find possible solutions in an existing environment. For example, the APIs discussed in Section 2.2.4 come from very different sources. While DOM originally was a company standard (invented by Netscape) and then was taken over by the W3C, SAX started as and still is an informal activity of a mailing list and thus interested individuals. So far, however, both specifications have been proven to be very useful to developers. While DOM shows some of the signs of committee work (an increasingly complex specification with many differently interested parties driving its further development), SAX remains the light-weight alternative to DOM it was designed to be.

Looking at the broad characterization of how technologies are defined in the markup language area, it can be said that the most important sources of markup language technologies come from the standard's bodies class. The following organizations are the most influential:

- *World Wide Web Consortium (W3C):* The W3C is a consortium which is concerned with standardization of Web technologies, and because the Web has gained so much influence in many areas, the influence of the W3C in turn has increased. Originally, the W3C has been founded by Tim Berners-Lee, the inventor of the Web, who was disappointed with the slow process of IETF standardization. Today, almost all companies who are actively interested in Web technologies are members of the W3C, and w3C membership in principle is open to everybody. In the markup language area, all core standards in the XML family of technologies have been standardized by the W3C, most notably XML itself, XML Namespaces, and DOM. Furthermore, many associated areas such as Web Services or the upcoming *XQuery* [7] query language for XML documents have been developed under the auspices of the W3C.
- *International Organization for Standardization (ISO):* ISO is concerned with many more standards than just information technologies, but there are many committees within ISO which are working on information technology standards. Technically, only ISO documents may be called "standards" (which is why W3C documents are called "recommendations"), because ISO is a world-wide organization which not only brings together companies and interested people, but has a very political balloting process. The most important ISO standard in the markup language area is SGML [22].

  ISO also decided to standardize some other languages such as an ISO version of HTML [25], but none has achieved the importance and acceptance of SGML.
- *Organization for the Advancement of Structured Information Standards (OASIS):* In contrast to W3C and ISO, which often tackle very fundamental areas, OASIS is more driven by immediate requirements from users of existing technologies, and their goal to build on a technology which has some level of recognition. One example for this is the often overlooked management of XML entity identifiers, which has been covered by OASIS with *XML Catalogs* [55]. OASIS is open for participation for everybody, and while many OASIS specifications are of interest

to rather small user groups only, this is one of the reason why OASIS exists and how the interests of these rather small users group can be served with an efficient and light-weight process for developing specifications which are not owned by a single company or individual.

- *Internet Engineering Task Force (IETF):* The IETF is the organization that defines all protocol-level standards for the Internet. Thus, in most cases its work concentrates on layers which are lower in the protocol stack that the ones required for electronic publishing. However, in some cases there are overlaps and some of the IETF's standards are definitely relevant for electronic publishing. The two most prominent examples are the *Universal Resource Identifier (URI)* [4], which defines the basic identification scheme for resources on the Web, and the *Hypertext Transfer Protocol (HTTP)* [20], which is the transport protocol of the Web. Both standards are visible on the application level (for example in URIs such as `http://www.ietf.org/`), and thus are relevant for many architectures of electronic publishing systems.

The above list of organizations is not complete, and it should be kept in mind that the descriptions of the organizations are concerning core technologies only, which means technologies which are generic enough to be of interest for potentially every markup language scenario. ISO and OASIS also define very many application-specific markup languages and associated technologies, but these are outside the scope of this list. The reason for this is that without the knowledge of a specific application area, it is impossible to make a statement about the influence that different organizations may have in this particular area.

2.5. Generic Markup Languages

Strictly speaking, SGML and XML are not markup languages, because they are defining how to use elements and attributes, but they do not make any statements about the elements and attributes being used. While SGML depends on the existence of a DTD (which thus could be viewed as defining "an SGML markup language", which is called an *SGML application* in SGML's own terms), XML can be used without a DTD, using *well-formed* XML only (see Section 3.3 for a definition of well-formed XML). XML thus widens the scope of how a concrete markup language (a set of elements and attributes and rules for how to combine them) can be defined. In fact, some popular markup languages are not based on DTDs, they use other mechanisms (other schema languages or even no schema language at all) for defining the vocabulary, more information about this subject can be found in Section 3.3.

If XML and SGML itself are not markup languages, but foundations for defining and using markup languages, the question remains whether the vocabulary defined by a concrete markup language is defined to cover a rather wide area of applications, or whether the application are is very focused and narrow. This is a typical case where two goals have to be balanced. A more generic markup language will be applicable in more cases, while it will lack the features that may be required in a very specific case. HTML is the prototype of a generic markup language. It is designed to represent "documents on the Web", and works reasonably well for more than 4 billion documents. Many Web designers and content

providers are probably wishing that it had some additional features, but since there are s many different users of HTML, the potential list of change requests is very large.
Interestingly, HTML has chosen a different path than constantly revising the language itself: Through declarative and procedural methods, HTML documents can be designed in ways which make them much more powerful that HTML alone allows. The declarative mechanism is the *Cascading Style Sheets (CSS)* [42] language. CSS allows HTML content to be presented in a way which goes far beyond HTML's rather simple formatting model. If declarative CSS is not sufficient, HTML content can be combined with procedural code written in *JavaScript* [26], which interacts with the HTML document and the Web user through the DOM API described in Section 2.2.4.
While HTML certainly is the most successful generic markup language, it lacks many features required for more sophisticates electronic publishing than Web publishing, and thus is generally considered inadequate as a generic markup language for electronic publishing. Thus, it should always be kept in mind that even though generic markup languages are designed to be generic, all of them will have their strengths and weaknesses. Choosing (or defining) a markup language for an electronic publishing project is one of the core tasks, and should start with a thorough requirements analysis and a survey of existing languages that seem to fit reasonably[3]. Also, HTML's approach of building extensibility into the language (through declarative and/or procedural mechanisms) should always be taken into consideration (Wilde [60] describes this approach, even though in a different application area).
Coming from different application areas, a number of generic markup languages have had some success, and are at least interesting candidates when looking for generic markup languages. These languages could also be used as a foundation for further refinement (through restricting or extending the language). The obvious advantages when choosing an existing generic markup language are existing experience reports and tools.

- *Text Encoding Initiative (TEI)* [49]: An international and interdisciplinary standard for representing literary and linguistic texts. The TEI standard is maintained by a consortium of institutions and projects worldwide (http://www.tei-c.org/). While TEI clearly has its roots in the humanities and related fields, it can also be used for other texts, as long as these do not require any application-specific markup not provided by TEI. TEI is not a single DTD, but provides an environment for creating customized DTDs. Therefore, TEI is very flexible and can be extended or restricted in many different ways.
- *ISO 12083* [24]: This ISO standard is a collection of 4 SGML DTDs for articles, books, serial publications (for example journals), and math formulae to be embedded in the first 3 DTDs. This standard has been developed mainly with the publishing industry in focus and gained some traction in the late 1990s. However, the standard itself is using SGML, and there never was an updated version of XML DTDs. Therefore, users of ISO 12083 were often switching to other, XML-based vocabularies (or developing their own XML-version of ISO 12083), which

[3] http://xml.coverpages.org/gen-apps.html lists a number of generic markup languages.

means that this standard can still be found in electronic publishing systems from the late 1990s, but newer systems will use other vocabularies.

- *DocBook* [54]: DocBook has been created for writing books and papers about computer hardware and software, but it is not limited to these application areas (it was started by publisher O'Reilly). As with TEI, it is possible to stretch DocBook to cover other application areas, as long as these do not require any special constructs which are not supported by DocBook. DocBook is available as XML and SGML DTD, and it is maintained by OASIS (http://www.oasis-open.org/docbook/). Because it is targeted at technical material, it is also often used for documentation of projects or products, and some big open source software initiatives maintain their documentation in DocBook. Not only can it be easily edited by the many contributors, because it uses XML and the DTD is fairly easy to understand, but also is it possible to generate PDF, HTML and help page versions of the documentation for a single source.

2.6. Application-Specific Markup Languages

The generic markup languages described in the previous section have the advantage of being applicable to a range of scenarios, the trade-off for which is that they are not tailored to fit one scenario as good as possible. Before discussing more specialized markup languages, it is useful to introduce a classification of markup languages which makes it easier to understand how tailoring a markup language to one scenario can be achieved.

- *Host Languages:* Markup languages which are host languages are designed to provide the frame for a document, the overall structure and general mechanisms how documents can be used. Host languages often are designed with openness and extensibility in mind, because it is clear that the framework alone will have to be extended to be useful in specific scenarios. The prototypical host language is HTML, which provides the overall structure for Web documents, and from its first version had the built-in rule that unknown element and/or attributes are not an error, but must be silently ignored by the application (i.e., the browser).
- *Embedded Languages:* When using a host language, the overall structure is well-known, whereas specific aspects of the application scenario may not yet be covered. A typical embedded language is the *Mathematical Markup Language (MathML)* [2], which is designed as a language for representing mathematical formulae, which in almost all cases will not be self-contained documents, but integral parts of documents. By embedding MathML markup within HTML (XML Namespaces as described in Section 2.2.3 are designed to support this kind of scenario, as shown in Figure 5), a document author can combine HTML's structuring capabilities for documents with MathML's expressive power for capturing mathematics. Of course, an application processing this document must support both languages, but extending an HTML browser to support MathML is much easier than writing an entirely new application to handle an entirely new document type for documents containing mathematical formulae.

With this separation of host and embedded languages in mind, it should become clear that for application-specific markup languages, it often is a good approach to take a generic

markup language and enrich it with the features that are required for the specific application scenario. Of course, this only makes sense if an appropriate generic markup language can be identified, otherwise it may still be necessary to create an entirely new application specific markup language. This is particularly true for scenarios where the principal datatype is not documents but something different, for which no generic markup language has been defined so far.

2.6.1. Textual Information

In case of textual information structured in document form, the basic content type is text, but markup is used to convey semantics of the text which go beyond unstructured text. Structural information about text may come from any application area, and in fact there are hundreds or maybe thousands of markup languages for very different types of texts, ranging from recipes to sermons. While many of these markup languages are only of interest for rather small user communities, some have been defined in contexts of larger commercial applications, and have become popular within an application domain. One such example is NewsML [30]

While the examples described so far are *host languages* in the sense describe above, there are also embedded markup languages for textual information. One such example is the MathML language mentioned above, which has been designed to represent mathematical formulae. Normally, mathematical formulae will never be complete documents, but appear within the context of (host language) documents. Using an existing embedded language in such a case has advantages in terms of existing vocabulary and possibly software, users which maybe are already familiar with the embedded language, and the forced separation of the markup language into different structuring mechanisms.

MathML is a very interesting example, because it actually combines both textual and non-textual representations. Through its so-called "presentation markup", a mathematical formula can be described by MathML in terms of textual (or symbolical) composition (e.g., the number '2' followed by the operator '+' followed by the number '2' make up the formula '2 + 2'), while through "content markup", a mathematical formula can be described in terms of mathematical concepts (the 'plus' function with the two decimal numbers '2' and '2', which in infix notation can be written as '2 + 2', but also could be written as '10 +10' in binary notation or as '2 2 +' in decimal postfix notation). While the presentation markup can be used to represent any mathematical formula that can be written down, the content markup can only represent the (rather small) set of mathematical concepts which have been included in the MathML content markup design.

Another variation would be to have an embedded language which adds non-textual semantics to textual information, such as the XML Linking Language (XLink) [16] (which is described in detail in Section 3.7.3). With XLink, it is possible to have a standard way of assigning hyperlink semantics to content, so that an otherwise non-hyperlink-enabled vocabulary can be extended to include hypermedia functionality. Instead of re-designing this functionality, the XLink vocabulary could be used. Again, the advantage of such a modular approach is the reuse of an existing standard vocabulary, the possible reuse of existing software for processing this vocabulary, and the fact that the overall design of the

application markup maintains a strict separation of different facets of the structures to be represented by markup.

2.6.2. Non-Textual Information

While textual information contained in documents is a very frequent datatype in electronic publishing, there are also many other (i.e., non-textual) datatypes which are relevant for electronic publishing, such as graphics and descriptive metadata. For textual as well as for non-textual information, the information is structured and thus could be represented using a markup language, but for the non-textual information, document-oriented markup languages will not provide the required features.

One important example for a non-textual markup language is the *Scalable Vector Graphics (SVG)* [19] format, which is used to represent vector graphics. Vector graphics is an image format which is based on graphic elements such as lines, rectangles, or circles, rather than the pixel-oriented format of raster graphics formats. The important advantage of vector graphics is their ability to scale well, i.e. they can be used in different output resolutions and small parts of an image can be magnified without getting the typical artifacts of raster graphics scaling. Consequently, vector graphics are an important format for electronic publishing, which typically has very different quality requirements depending on the output medium (for example, the resolution of a computer screen is very different from the output resolution of a high-quality document imaging system).

Traditionally, programs used for producing and editing vector graphics used proprietary file formats, on the one hand because there was no common standard which could be used, and on the other hand because each program had a slightly different set of features, and the file format reflected this set of features. However, increasingly programs start to write and read standard formats such as SVG, too. For users, this means that vector graphics images can be shared by different programs, which is very useful. However, because the feature set of many applications is bigger than that of, for example, SVG, working with this standard format has the disadvantage of not being able to use some of the special features that are provided by vector graphics programs. However, from the point of a user who is interested in a portable format which also is stable enough to be used for archiving, it makes a lot of sense to only use the features supported by the standard format.

Another important class of non-textual markup is the area of metadata. Metadata simply is data about data, which means it is data describing data. Usually, this description is structured using some vocabulary, making assertions about the described data (such as the author and the creation date). Thus, one important aspect of metadata is that it is usually highly structured, and this structure can be represented using markup. The whole area of metadata has received a lot of attention in the context of the *Semantic Web* [5] activities, which basically are an attempt to describe data on the Web in a way which enables machines to “understand” the data.

In the Web context, the *Resource Description Framework (RDF)* [36] has become the most widely used approach for representing metadata, and even though RDF itself is an abstract model, the by far most widely used way to represent RDF data is using its XML syntax [3]. In this case, the markup representation of the information is not required by the information model behind it, but experience has shown that it is convenient to use a markup language

representation, which can be generated and interpreted using standard XML tools. Of course, after parsing the XML, it is still necessary to check the resulting structures whether they indeed represent valid RDF data. This has emerged as one of the disadvantages of the approach of encoding RDF in XML: There are so many ways to encode the same RDF data in different XML representations, that it is confusing for developers and sometimes hard to manage. This is an interesting example of markup languages as a two-sided sword: It may be convenient to use them because of the availability of tools, but they may very well not be the ideal representation for every type of information. Which leads to the next topic, non-markup approaches to information representation.

2.7. Non-Markup Approaches

Non-markup information representation is a very wide field, and the vast majority of data today is represented in non-markup formats, even though the success of HTML and XML has made markup a widely used technology. For many application areas, using markup is not an option, because of the amount of information and the requirement to store data in a compact form (markup is useful and easily usable, but it definitely is not a compact format). Typical application areas with non-markup formats use data that is sampled, for example raster images, audio, and video (a sequence of raster images). In these application areas, i would not make any sense to use markup to represent data, and the data formats usually are very compact and optimized to not waste any space with redundant information (which is abundant in markup formats through the repeating occurrences of element and attribute names).

In some cases, existing data formats have been retrofitted with a markup language, such as *Microsoft Word*, which in its newest version uses an XML-based data format. In principle, it is not a technical problem to define a markup format for any non-markup data format, but the interesting question is whether there is any benefit in doing so. Another question is whether using a markup format may actually lose some important features of an existing format. As an example, Adobe states that is would be willing to migrate the *Portable Document Format (PDF)* [1] to an XML foundation, but only if there is a more compact form of XML, and only if it is possible to directly address parts of a document without having to parse the entire contents before this part. So far, Adobe has only decided to embed markup-based metadata in the PDF using the *Extensible Metadata Platform (XMP)* format, but this is only a very small subset of the PDF format.

Generally, many application areas could benefit from a more compact representation of XML, which in lack of a better term is often referred to as “binary XML” (which is a contradiction in itself because a markup language is not binary). Starting with a workshop in the fall of 2003, the W3C has formed a working group working on the requirements [21] and use cases of binary XML, but at the time of writing there is no concrete format that has been proposed. However, it is a general consensus that there is a need for a binary XML, and in order to prevent a fragmentation of the XML landscape, the has W3C decided to produce a binary XML recommendation. When this specification is available, many more application areas than today will be able to switch to a markup-based data model, even if the encoding may use a compact binary format.

3. Structuring Content

Section 2 introduces markup languages as a way of representing document structures, which are always tree structures (in the popular markup languages SGML and XML). In the following sections, the questions of how given information should be structured (Section 3.2), and how a class definition can help in validating content against a schema (Section 3.3 and following sections) are discussed. However, as a preliminary discussion the difference between markup and content has to be discussed, which is of great importance for virtually all XML technologies in use today.

3.1. Markup vs. Content

Markup is a way of representing a structure through text-based encoding. Therefore, markup languages always need some special characters representing the structure, while the non-markup characters are considered to be normal content. However, virtually all markup languages allow a certain degree of variation in the syntax, the simplest being whitespace in tags. For example, the markup `<x a="1" />` and `<x a='1'/>` (with space characters shown as for better visualization) in most cases will be considered as being equivalent, even though it is not identical. In this case, a minor variation in markup does not affect the structure hat is being represented.

One step further, variations of some markup characters are also considered insignificant by most applications, for example the two elements `<x a="1">` and `<x a='1'>` are both simply `x` elements with an `a` attribute having the value 1, and the fact that the value is delimited by a different type of quote does not change this basic interpretation of the markup. However, it should be noted that this is a simple convention that is accepted by the majority of markup language users, it is not a feature of the markup language itself. SGML and XML simply allow different delimiters to be used for attribute values[4], and since this is a simple syntax issue, it is left open whether the different delimiters carry different semantics.

Going one step further, most SGML and XML users will agree that `<x a="1" b="2">` and `<x b="2" a="1">` are equivalent (an `x` element with attributes `a` and `b`), because attributes are often treated as having no order. Again, this is a simple convention, mostly established through typical usage and the fact that neither SGML nor XML schema languages exist that provide support for enforcing attribute order. It would be possible to define an application scenario where attribute order was significant (or attribute delimiters, or whitespace in tags), but it would not be wise, because it would prohibit the usage of standard SGML or XML tools, which typically do not report these syntactic variations to the application.

In order to make clear which parts of a markup language are “real structure and content” and which are “just markup”, it is necessary to define a data model, which clearly identifies the relevant parts of the content being represented by markup. In case of XML, this data

[4] Through it separation of an abstract and a concrete syntax, SGML allows virtually all markup characters to be chosen, while XML is based on a character-based syntax and simply allows some minor variations (such as the quotes delimiting attribute values).

model is the *XML Infoset* [15]. Looking behind the scenes, it is interesting to note that more or less all relevant XML technologies in fact are not XML technologies in the technically strict sense, but Infoset technologies. XQuery [7], XSLT [32], SOAP [43] and the DOM/SAX APIs all are based on the Infoset in their latest versions. This makes XML simply a syntax for encoding (and exchanging) Infosets, with all the processing being performed on the abstract model.

While for most purposes, the difference between XML and the Infoset may seem negligible, it is important to keep in mind that in a markup-based processing model, the actual processing of content should be based on the underlying data model, rather than the textual markup data itself. This way, users can be sure that their applications are supported by tools working with the data model.

3.2. Identifying Structures

The question of how to represent structures with markup as discussed in the previous section is very important, because it shows some limitations of how popular markup languages are being used (e.g., attribute order is insignificant). However, another question is at least equally important, and this what structures should be represented in the first place.

As a general rule, all structures which should be accessible to later processing stages should be made available through markup. For example, if there is or will be a need to process citations in a text, they should not be entered as text-based structures (such as `[Berners-Lee89]`), but as markup (such as `<cite ref="Berners-Lee89"/>`). All structures which are not available as markup must be extracted using text-based processing, which requires programming efforts and in most cases introduces ambiguities and other problems which could be avoided using markup.

After identifying the structures which should be accessible to later processing stages, it is also important to consider the granularity of the information. For example, if it is required that individual sections of a documents can be marked with an author's name, it would make sense to use markup to identify this name. However, there still remains the questions of how this markup should be designed, and the following list shows some possible approaches to represent a name using markup:

`<name>Philippe C. Cattin</name>` — In the simplest solution, the name could be represented as an element which contains the full name. This approach is sufficient if the name is only used as a string in all processing stages using the document. However, in many cases, names need to be processed in a more detailed way, in particular in a way which enables the separation of given and last names. This is often required to create lists of sorted names, where the sorting has to based on the last name. Consequently, structuring a name into its individual parts often is useful for some processing stages using the document.

`<name given="Philippe C." last="Cattin"/>` — One possible way to create a name with a separation into a given and a last name is to use

attributes to represent these to parts of a name. In principle, this is a viable solution, but there is one limitation of SGML and XML which may make this approach problematic: attributes cannot be repeated, so if there is a requirement to be able to capture different given names in individual structures, this cannot be achieved by using two `given` attributes (and since attributes often are considered as having no order, this would be problematic anyway). However, this could be circumvented by introducing a new structural component for a middle name, which for example is a very common concept in the U.S.

`<name given="Philippe" middle="C." last="Cattin"/>` — By introducing a new attribute `middle` for middle names, the name's structure can be represented more accurately. However, this model is very tightly bound to the view of names as consisting of one given, one middle, and one last name. To make this model more flexible (and thus better able to deal with other concepts of name structures), it would still be preferable to have the given name as a component which can be repeated, for example for names having more than just one given name. With this consideration in mind, it seems that attributes are not the best way to represent these structures, and it makes more sense to switch to elements (which can be repeated).

`<name><given>Philippe</given><middle>C.</middle><last>Cattin</last></name>` — A simple reformulation of the previous approach, using elements instead of attributes. However, this approach is more flexible, because now the `given` element could be used repeatedly, if a name needs to be represented which has more structural parts than the single given/middle/last parts facilitated by the attribute-based solution. However, seeing that a repeatable `given` element conflicts with the `middle` element (a middle name is a given name), it probably makes sense to eliminate the `middle` element and simply use repeatable `given` elements.

`<name><given>Philippe</given><given>C.</given><last>Cattin</last></name>` — This last example replaces the `middle` element with a `given` element, and shows that the `given` element is repeatable. This model better fits the view of a name as a sequence of given names and one last name, and if it absolutely required to identify the middle name (in its pretty limited U.S. sense), then this could be achieved by an attribute of the `given` element, identifying one of the given names as the middle name (`<given middle="yes">C.</given>`).

This sequence of models of representing names illustrates that even a simple example like a name can be modeled in a variety of ways, and that most of these ways have specific

limitations and advantages, which should be taken into account when choosing a model. It should also be noted that the given/last name concept shown as the last variation still is not an ideal solution, because is reflects the occidental structure of names, which in other cultures is not present and can lead to mapping problems (for example, Indonesian names often lack a last name and thus often cause problems when a name model with cultural bias has to be applied).

As demonstrated by this simple example, defining which structures should be represented by markup is a two-fold problem. Firstly, it has to be decided which structures of the data to be captured should be represented. This first problem is not so much a markup language problem, but a general data modeling problem. However, certain properties of popular markup languages should already be taken into consideration, such as the fact that markup is tree-structured. If the data model uses overlapping markup, it will be hard to map onto tree-structured markup languages.

Secondly, the data model must be mapped onto markup structures. If the data model respects the basic limitations of tree-structured markup languages, this mapping should not be a principal problem, but there still are many ways to map one data model onto concrete markup structures. For example, the choice between attributes and elements is one example, and while certain properties of the data model may make this choice clear (if a structure must be repeatable, it can only be represented by an element), in other cases the choice can be less clear and must be made by the designer of the markup. Vitali et al. [53] describe different design patterns for designing document structures.

When designing the markup language for a given data model, the markup language designer creates a markup vocabulary and rules describing how to use it, for example which elements or attributes are mandatory or optional, and which elements are repeatable. These rules are often called a document class, because they describe what documents should look like.

3.3. Document Classes

A document class captures the rules that any document claiming conformance to this class must adhere to. Conceptually, a document class could use any kind of rules, and in Sections 3.4, 3.5, and 3.6 a number of different technologies are discussed. In reality, a document class must be defined with rules which are written in a well-defined language, and in most cases, these languages are grammar-based, which means that they define rules how documents have to be constructed to conform to the class. Grammar-based approaches have two major advantages:

- *Similarity to Natural Languages:* Grammars are a well-known mechanism for describing languages, and the rules for a document class basically are a language for constructing documents of this class. When dealing with structured information, grammars often play an important role, in particular in the document/language domain. Thus, representing the rules as grammar allows users to apply their common knowledge about grammars in general.
- *Document Generation:* A grammar in generative in the sense that it can be used to construct a document from the grammar rules. Using this property of a grammar, the grammar's rules can be used to control a user interface for constructing

documents, and indeed many editing tools exist which are grammar-driven, so that users are guided by the system when entering content. This way, constructing documents of a given document class is much easier, because writing and checking are not two separate processes, but integrated into one grammar-guided writing process.

Thus, choosing a grammar-based approach for describing document classes is a popular decision, and therefore markup language technology is focused on this kind of document class description.

As described in Section 2.1, XML defines two rather different things, on the one hand it is a syntax for marking up documents using elements and attributes and some other constructs, and on the other hand XML defines a language for defining classes of XML documents, the *Document Type Definition (DTD)*. This has more historical than technical reasons, because SGML also defined both documents and DTDs, and XML was defined as a stripped-down version of SGML. Nowadays, most XML experts would probably prefer to have two separate standards for documents and DTDs, because DTDs are only one way of defining document classes, and they are gradually being replaced by other, improved mechanisms (such as XML Schema). However, being at it is, DTDs are hardwired into XML, and every implementation claiming XML conformance must implement at least some of the DTD mechanisms.

In the area of documents and DTDs, the biggest difference between SGML and XML is that XML allows documents which do not conform to any DTD, whereas SGML requires that for each document there must be a DTD. This seemingly small difference was one of the biggest factors of XML's success, and also is the reason for XML's ongoing evolution. Since XML does not require documents to conform to a DTD, it is possible to work with XML documents alone, ignoring the DTD mechanism. This is the reason why XML is often called to be "self-describing", in the sense that in the case of DTD-less documents, all information that is available for the interpretation of the document is the document itself.

The formal separation of DTD-based documents and DTD-less documents has led to the definition of two separate concepts, which are a aspect part of XML and document classes:

- *Well-Formed XML Documents:* A well-formed document conforms to the requirements of the XML specification, which means that it satisfies all syntactic requirements of the XML specification. In addition, it must satisfy the *well-formedness* constraints of the XML specification, for example start- and end-tags must be balanced.

  Basically, well-formedness of an XML document guarantees that it is a syntactically correct and thus can be interpreted as an XML documents. In practical terms, well-formedness assures that a document can be processed with standard XML tools, for example that it can be processed by an XSLT stylesheet or through a DOM- or SAX-based program.
- *Valid XML Documents:* If an XML document is well-formed, associated with a DTD (through a document type declaration as described in Section 3.4), and conforms to the constraints expressed by this DTD, the document is said to be valid with respect to this DTD. XML processors come in two flavors, non-validating and validating XML processors. Only validating processors must be able to check a document for validity, while non-validating only check for the

document for well-formedness.
In practical terms, validity guarantees that a document satisfies constraints which have been defined in a DTD, which means that it is a valid instance of the grammar defined by the DTD. Thus, validity makes it easier to define processing for XML documents, because applications processing the XML documents can be sure that the requirements of the DTD have been satisfied, and thus do not have to be checked. In most cases it is much easier to write a program which processes a validated XML rather than any XML, because the constraints in the DTD have been checked, and much less checking has to be performed within the application code.

When XML was created, DTDs were the only schema language available for XML, but in principle the idea of a schema language is not limited to one specific language. Conceptually, a schema language is used to define document class, documents conforming to the schema language are instances of the class, other documents are outside of this class. The advantages of this descriptive approach compared to application code checking for conformance are a portable and platform-independent definition of a document class, and reduced coding efforts for application developers.
The concept of document classes is simply based on the separation of XML documents into valid and invalid documents. In order to implement document classes, they must be described, and the common term for describing document classes is that of a schema language. DTDs are not the only *schema language* anymore, and the following section briefly describes the development and future of XML schema languages.

3.3.1. Schema Languages

A document class is a concept and can be defined in any way suitable for a given application scenario. It may be described informally (using plain text) or formally. Formal definitions may be based on any applicable declarative approach, or procedurally, where a program processes documents and the result of this processing answers the question whether a given document conforms to a document class or not. In many cases, however, it makes sense to describe a document using a schema language, which is a declarative language for describing which documents belong to a document class, and which do not. In case of XML, it can be assumed that a document has to be well-formed XML at the very least, and starting from this lowest common denominator, a schema language enables applications to check whether a well-formed XML document also belongs to a given document class.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE document SYSTEM "document.dtd">
<document created="20050226" modified="20050315">
    <author>Erik Wilde</author>
    <title>Markup Languages</title>
    <p>Non-linear content (as discussed in the previous section)
       and structured content in general needs to be
       represented...</p>
```

```
    <p>In Section <xref id="using-markup"/>, it is explained
       how...</p>
    <section id="using-markup">
       <title>Using Markup Languages</title>
       <p>After this short history of markup languages, we will
              now look at how to use markup languages...</p>
    </section>
</document>
```

Figure 8: Document Type Declaration

A very important aspect of schema languages goes beyond mere validation. While validation is a useful concept for determining whether a given document conforms to a document class or not, a schema language can often be used to guide the process of document creation (this applies mostly to grammar-based schema languages such as DTD and XML Schema, while other classes of schema languages ar less suited for this purpose). When using authoring systems, it can be very helpful for users if the system provides immediate feedback whether a document os valid or not. An authoring system may even refuse to make any changes which would render a document invalid. Thus, a schema language can vastly improve the quality of authored documents, if it is being used as the foundation of an authoring system.

Strictly speaking, the terms *validation* and *valid document* are limited to DTDs only, which are described in Section 3.4. Conceptually, however, the terms *validation* and *valid document* are often used in a more general sense, which simply refers to the fact that an XML document is validated against a schema language to determine its validity.

The most important XML schema language today is XML Schema described in Section 3.5, which is rapidly becoming the successor of DTDs. The reason for this is that XML Schema has been created by the W3C, is supported by all major vendors of XML-oriented software, and is the foundation for other important XML technologies, such as *XSL Transformations (XSLT) 2.0* [32] and *XQuery* [7].

Even though XML Schema probably will be the most important XML schema language for the following years, it is not the perfect fit for all application scenarios. In some cases, it is too complex, in others it lacks features which are considered essential. In these cases, it makes sense to look at other schema languages as described in Section 3.6, which may be useful as a complement or replacement for XML Schema.

3.4. Document Type Definition (DTD)

The DTD schema language is the oldest and most widely known schema language for XML. The reason fore this is that it is an integral part of the XML specification itself. While XML processors are not required to support DTD validation (XML allows *validating* and *non-validating* XML processors), most processors are validating processors, and thus no special software is required for using DTDs.

As described in Section 3.3, XML documents may be either *well-formed*, which means that they are syntactically correct XML documents, or they may be *valid*, which means that they also adhere to the constraints defined by a DTD. When an XML processor parses a document, it uses the *Document Type Declaration* to determine the DTD associated with this document.
Figure 8 shows the example from Figure 3, but this time with a document type declaration, the `DOCTYPE` line in the document. A document type declaration may use a *system* or a *public* identifier for the document class, in both cases it provides information for the XML processor to identify and/or locate the DTD. In this example, the document type declaration uses a system identifier and a simple file name (technically, this is a relative URI), which means that the `document.dtd` file has to be in the same directory as the XML document.
In most cases, DTDs are either located through an absolute URI, or only through their public identifier. How public identifiers are mapped to actual locations is a problem that is outside of the XML specification, but has been addressed by the specification of *XML Catalogs* [55].
If the XML processor is able to locate the DTD, it retrieves the DTD and checks the XML document against the constraints (which define the document class) defined by the DTD. Figure 9 shows an example DTD for the document shown in Figure 8, it is a possible way of how a document class could be defined.

```
<!ELEMENT document  (author?, title, p+, section*)>
<!ATTLIST document
          created   CDATA #REQUIRED
          modified  CDATA #IMPLIED >
<!ELEMENT author    (#PCDATA)>
<!ELEMENT title     (#PCDATA)>
<!ELEMENT p         (#PCDATA | xref)*>
<!ELEMENT section   (title, p+)>
<!ATTLIST section
          id        ID    #IMPLIED >
<!ELEMENT xref      EMPTY>
<!ATTLIST xref
          id        IDREF #REQUIRED >
```

Figure 9: Sample XML DTD

Basically, a DTD defines rules for how valid documents have to be structured; in essence this is a grammar. The rules are defined for elements, defining the allowed contents of an element (in the `ELEMENT` construct, called an *element type declaration*), and the allowed attributes of an element (in the `ATTLIST` construct, called an *attribute-list declaration*). The syntax of a DTD is slightly different from that of an XML document, even though

some characters (most notably, the angle brackets) are the same. Interestingly, a DTD is not an XML document, which makes it harder than necessary to process DTDs.

The most important part of a DTD are the element type declarations, because they specify the allowed content of elements. Since only elements may contain elements, this is the part where the complexity and richness of XML is available, because elements may contain themselves directly or indirectly, creating recursive definitions within a document class (theoretically, this allows XML documents of any nesting level, which of course will be limited by the actual nesting levels accepted by XML implementations). Also, elements may be reused, such as the `title` and `p` elements in the example DTD, which are allowed within `document` and `section` elements. The document element of a document class (i.e., the outermost element of the documents) is not explicitly marked, thus the document class defined by the DTD also includes subtrees of the original document trees.

Element type declarations specify an element type's content. As a special case, element types may be declared to be `EMPTY`, which makes sense if they have attributes (like the example's `xref` element), or if the occurrence of an element alone carries all information and does not require any additional information. In most cases, element types allow content, and the two basic types of content are *element content* and *mixed content*. Element content means that an element may only contain elements, i.e. no character data (other than whitespace). In the example DTD, the `document` and `section` element types are defined as having element content. Generally, element content can be defined using the following mechanisms:

- *Sequences:* This specifies that an element type may contain a sequence of content. Sequences are defined by using a ',', which separates the individual items of a sequence.
- *Choices:* If content has to be defined as a choice between several alternatives, the alternatives are specified and separated by a '|'. Such a choice means that any of the alternatives is accepted when occurring in an XML document, but only one of them may occur.
- *Optional Parts:* If items are optional, they are followed by a '?', indicating that they may or may not occur in an XML document. In the example, the `author` element is indicated as being optional, so documents may or may not specify the author's name in an `author` element.
- *Repeatable Parts:* If items can be repeated, they are followed by a '+', indicating that this item must occur, but may be repeated.
- *Optional and Repeatable Parts:* If items are optional and can be repeated, they are followed by a '*', indicating that this item may occur, and may be repeated.

These mechanisms can be freely combined in the specification of an element type's content, and parentheses can be used for grouping, which allows very complex content models, such as `"((a|b)?,(c|(d,e)*)+)"`. In practice, however, most content models are not very complex expressions, but rather simple, as shown in the example.

The other basic type of an element type's content is mixed content, which means that the element may contain elements as well as characters. In the example, the `p` element type

uses mixed content, specifying that `p` elements may contain any mixture of character data and `xref` elements. The `author` and `title` elements also use mixed content, but in the very simple form of character data being mixed with no elements. In all these cases, the mixed content is indicated by using the keyword `#PCDATA`, which represents the character data in the content model.

Mixed content always has the form "`(#PCDATA | element | ... )*`", with the elements allowed in the mixed content being listed as parts of the overall choice. It is not possible to restrict the number of occurrences of these elements, or to force a certain sequence of elements within mixed content. The mixed content model looks as if it could be varied (and in SGML this was actually possible), but XML only allows this form of mixed content, with everything being within one choice which is marked as optional and repeatable.

In addition to element type declarations, a DTD also contains attribute-list declarations, which specify the attributes which are allowed for element types. An attribute-list declaration specifies the element type for which it defines the attributes, and then lists all attributes. By definition, attributes are defined as a set, which means that they don't have to occur in any specific order in the document. Each attribute is defined by a name, a type, and a default declaration.

As an example, the `document` element has an attribute-list declaration associated with it, which specifies a required (`#REQUIRED`) `created` and an optional (`#IMPLIED`) `modified` attribute. Both are defined as having the string (`CDATA`) type, which means that they may contain arbitrary strings.

Apart from the string type, XML supports a number of *tokenized types* for attributes, which restrict the attribute values syntactically and add semantics to the attribute value. The most important tokenized types are `ID` and `IDREF`, which turn attributes into identifiers respectively references to these identifiers. As shown in Figure 8, the `ID/IDREF` mechanism makes it possible to identify and reference sections within a document, and by using the `ID/IDREF` attribute types, the XML processor recognizes the identifier and reference semantics of the attributes and is able to check the document for consistency.

Specifically, for `ID` attributes, the XML processor checks that no two attributes of this type have the same value, otherwise the document is invalid. For `IDREF` attributes, the XML processor checks that each `IDREF` attribute contains a value of an `ID` attribute, otherwise the document is invalid. This makes it easy to include consistency checking of identifiers and references as part of the document class, rather than checking it on the application level.

In addition to the element type and attribute-list declarations, XML DTDs support a number of other mechanisms, such as parameter entities and conditional sections. For a more detailed description of DTDs and how to use them in real-world scenarios, the excellent guide by Maler and El Andaloussi [39] provides in-depth information (it actually is about SGML, but can be used for XML as well).

3.4.1. Limitations of XML DTDs

While DTDs have a long tradition in their SGML form, and have also been very popular when XML appeared, they have a number of limitations, some of which were apparent from the very early days of SGML, and some of which became apparent when XML was used in a number of application areas for which it had not been designed. When discussing the limitations of XML DTDs, they can be grouped into principal limitations of the SGML DTD model, and in limitations which have been introduced when XML was designed as a stripped-down version of SGML.

The principal limitations of the SGML DTD model have been introduced when SGML was created in 1986:

- *Global ID/IDREF:* The concept of identifiers and references is global for the document type, which means that it is not possible to create identifiers for specific element types, or to create references to specific element types. In practice, this means that it is not possible to constrain identifiers and references in the way in which they are most often used in applications: as specific for the objects being identified or referenced.
- *Limited Datatypes:* While SGML has a bigger repertoire of attribute types than XML, it still lacks the attribute types which are often required from the application point of view, such as dates or times. SGML's attribute types are very markup-centric and provide only little support for application-level attribute types.
- *Weak Support for Reuse and Versioning:* In SGML, element type declarations are always global, and attribute-list declarations are always local. In practice, many of SGML's weaknesses in the area of how parts of a DTD can be reused for other DTDs or for versioning, have been addressed by design pattern involving parameter entities, but this was never more than a makeshift solution.
- *No Namespace Support:* XML Namespaces (as described in Section 2.2.3) have been introduced as a separate specification after SGML and XML already existed, and thus it is inappropriate to ask for namespace support in DTDs. However, XML Namespaces have become a very important piece of the XML landscape, and supporting the separation of names into a prefix and a local name is rather cumbersome with DTDs.

While these principal limitations have been introduced by SGML, when the concept of DTDs was standardized, the following restrictions have been introduced by XML's stripped-down version of SGML DTDs (as described in Section 3.4):

- *Limited Mixed Content:* XML only allows mixed content with the choice operator and any number of repetitions, so there is no way to further constrain mixed content. In SGML, the `#PCDATA` could be used more or less like an element within any kind of content model, which allowed more control over mixed content.
- *No Inclusions or Exclusions:* SGML's exceptions were a mechanism to define inclusions or exclusions of elements in a element type declaration. This affected the whole subtree within an element, not only the immediate children of an element (which makes exceptions very powerful, but also hard to handle). XML

sacrificed exceptions because they were considered too complex, and were not used in many DTDs because of their side-effects anyway.

- *No AND Groups:* SGML's AND group (using the operator '&') was a third way of combining elements in a content model, next to sequences and choices (called OR in SGML). The AND group specified that all items connected with this operator should occur, but that the sequence did not matter. Most content models using the AND group can be replaced by XML DTD constructs [33], but for some content models this is impossible, and in general it leads to very long content models.

Fueled by these limitations of XML DTDs and many XML users who required a more powerful mechanism, a new schema language was developed by the W3C, starting with a variety of proposals from individual players in the software industry *lee00*, most notably Microsoft's *XML Data* and *XML Data Reduced (XDR)* languages, and Commerce *One's Schema for Object Oriented XML (SOX)*. The result of W3C's work was called XML Schema, and because of its endorsement from the W3C quickly was accepted by all major players, getting ahead of all other proposed schema languages in popularity.

3.5. XML Schema

XML Schema [51, 6] is a much more complex schema language than DTD, and while this also opens new opportunities, it also makes it harder to learn the schema language, and to use it. The two major differences to DTD are the following:

- *Structural Types:* XML Schema introduces a type layer, which means that elements as well as attributes are not declared by specifying their contents directly. Instead, types are being used, with XML Schema introducing *complex* (for elements only) and *simple* (for elements and attributes) types. Types can be declared with names, reused, and can be derived from other type through various type derivation mechanisms.
- *Datatypes:* XML Schema introduces a library of *built-in* simple type datatypes, which can be used or derived for often used concepts such as URIs, dates, times, or durations. This is a huge improvement over DTDs, which did not have any application-oriented datatype concept and for example did not even allow to specify that an attribute value had to be a number. The type derivation mechanisms described above can be used to derive own types from the built-in types, for example defining a number type with lower and upper bounds.

Apart from that, another immediately notable difference is the syntax. DTDs use a syntax which is not XML document syntax, while an XML Schema is an XML document (which leads to the consequence that there must be an XML Schema for XML Schema, which actually exists, which is the description of the XML Schema XML syntax using XML Schema).

The example XML Schema shown in Figure 10 is almost equivalent to the DTD shown in Figure 9. It is an XML document, using XML elements and attributes for defining the schema. However, elements are not defined by simply defining their content model and attributes, they are defined by defining complex types, which in turn contain the content

models and attributes. The XML Schema shown in Figure 10 is only almost equivalent to the DTD shown in Figure 9, because it is more restrictive, limiting the attribute values of the `created` and `modified` attributes to be of type `xs:date` (one of XML Schema's built-in types), constraining them to contain only valid dates, something which was not possible within the DTD.

While the XML Schema shown in Figure 10 is structurally equivalent to the DTD, it is also much longer and harder to read, which mostly is caused by the XML syntax. An alternative syntax for XML Schema which is easier to use for humans is the XML Schema Compact Syntax (XSCS) [57], which makes it easier to read and write XML Schemas. An alternative to the compact schema syntax are graphical schema representations, one example is shown in Figure 11.

There is no standard or specification for graphical schema representations, most of them are provided by XML tools (such as XML editors or XML Schema designers) and can be used for display as well as for editing.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="document">
    <xs:complexType>
       <xs:sequence>
          <xs:element name="author" type="xs:string" minOccurs="0"/>
             <xs:element ref="title"/>
          <xs:element ref="p" maxOccurs="unbounded"/>
          <xs:element name="section" minOccurs="0"
             maxOccurs="unbounded">
                <xs:complexType>
                   <xs:sequence>
                     <xs:element ref="title"/>
                     <xs:element ref="p" maxOccurs="unbounded"/>
                   </xs:sequence>
                <xs:attribute name="id" type="xs:ID"/>
                </xs:complexType>
          </xs:element>
       </xs:sequence>
       <xs:attribute name="created" type="xs:date" use="required"/>
       <xs:attribute name="modified" type="xs:date"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="title" type="xs:string"/>
  <xs:element name="p">
     <xs:complexType mixed="true">
          <xs:choice minOccurs="0" maxOccurs="unbounded">
             <xs:element name="xref">
               <xs:complexType>
                 <xs:attribute name="id" type="xs:IDREF"
                           use="required"/>
               </xs:complexType>
             </xs:element>
          </xs:choice>
     </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 10: Sample XML Schema

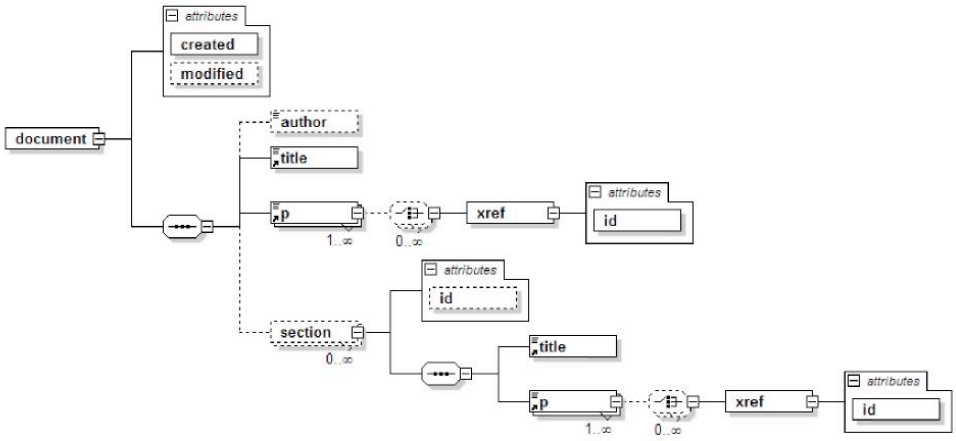

Figure 11: Graphical View of XML Schema

Apart from the most fundamental features and improvements in comparison to DTDs described above (structural types, datatypes, and XML syntax), XML Schema also has the following features:

- *Namespace Support:* XML Schema supports XML Namespaces (as described in Section 2.2.3) by allowing a schema to specify its target namespace, and by allowing a schema to refer to other namespaces through wildcards which only allow elements and/or attributes from specified namespace(s). This makes it easier to create modular vocabularies which can be combined on the namespace level.
- *Identity Constraints:* DTDs has the concept of ID/IDREF attributes, where IDs are guaranteed to be unique within a document, and IDREFs are guaranteed to reference an existing ID. These are both global concepts, so it is impossible to reuse the same ID value on different elements. XML Schema introduces identity constraints as a generalized ID/IDREF concept, which decouples constraints from the type of the values, makes it possible to define scoped constraints, and supports multi-field and element-based constraints.
- *Improved Model Groups:* The sequence and choice model groups of XML DTDs are complemented with a all model group, which allows the specification of parts which must occur, but in any order.[5] As another improvement, XML allows more specific mixed content, which means that is possible to specify for example a sequence of elements in a mixed content model, and these elements must then occur in this sequence within the mixed content.
- *Improved Reuse of Schema Components:* XML Schema introduces *named model groups* and *attribute groups*, which both are ways to group and name parts of

[5] Unfortunately, the all model group has a number of restrictions which make it hard to use, and thus it is not seen very often in XML Schemas.

complex types, which can then be reused in different contexts. These features help to build better schemas by defining and using reusable components.

- Type Annotations: XML Schema is used for validating documents, but it is equally important for the type annotation of documents. This means that an XML Schema can be used to augment an XML document with type annotations, which can then be used in later processing stages for type-aware processing of the document's contents. This is defined by the *XQuery 1.0* and *XPath 2.0 Data Model* [18], which defines a data model for XML documents on the foundations of the XML Infoset and XML Schema.

While XML Schema has added new features to the previously rather simple world of XML schemas using DTDs, it has also introduced new complexity, and in particular a whole new modeling layer through the structural types using complex and simple types. This means that writing schemas has become more complex, and experience has shown that many users of XML Schema are not fully aware of language's complexity and intricacies, and thus do not use the language in the best possible way. In particular, many XML Schemas are structurally equivalent to DTDs, and simply use the XML Schema formalism because this is required by the environment.

Using XML Schema is not easy because of its complexity, but i can considerably improve working in an XML-centric environment because it provides a better document class definition than a DTD, provides data binding support through the use of well-designed structural types, and type annotations for supporting type-aware XML processing environments. An exhaustive description of XML Schema and how to use it has been published by van der Vlist [52], in the following sections the two most important areas for using XML Schema for electronic publishing are discussed briefly.

3.5.1. XML Schema Design Patterns

The principal question of how to design the markup that appears in XML documents has been discussed in Section 3.2. This question is independent from any particular schema language, because it deals with the question of how documents are supposed to be structured.

The question of how to implement this markup design with a particular schema language, on the other hand, clearly depends on the selected schema language, and on the features of this language. In case of XML Schema, this question of schema design is particularly important, because XML Schema has many different features to choose from, and thus any given markup design can be implemented with very different XML Schemas.

One important question is whether types and element and attribute definitions should be made reusable, or whether it is acceptable to use them only locally. In the schema shown in Figure 10, for example, all types are local (they have no name and thus cannot be reused), and most elements are local (defined within content models), but the element `document` is global (defined globally and then `ref`erenced) because it has to be (it is the document element), and the `title` and `p` elements are global because they are reused in different contexts (in `document` and `section`).

When defining XML Schemas, for many declarations (mostly types and element/attributes) it has to be decided whether they should be global (and thus reusable) or local (defined

where they are used and thus not reusable). While global declaration enables reuse, local declarations make the schema more compact, easier to handle, and "hide" the local components from outside views. There are different styles of how to combine local and global declarations, and the most popular style has been coined as Venetian Blinds style and is characterized by global type declarations and local element/attribute declarations. This style of schema designed is based on the assumption that reuse should be based on types, and not on elements/attributes. This becomes even more important in environments where the XML is processed with the type information being available, such as the XQuery [7] or XSLT 2.0 [32].

Another important issue in XML Schema is the question of how the reuse of components should be modeled at all. While DTDs provide no well-supported concept for reuse, XML Schema supports several, and the most important question in this regard often is whether to use reusable groups or type derivation. In both cases, components are reused, in the first case these are fragments of types, in the second case these are complete types, which can then be restricted or extended using XML Schema's different type derivation mechanisms. The question which way is preferable cannot be answered generally, it depends on a variety of factors such as intended reuse, preferred modeling style, and possible relations with models other than XML Schema with which the schema should be aligned.

It is impossible to answer general questions about designing XML Schemas on a couple of pages. The most important reason for this that the structural types introduced by XML Schema make it more powerful as a modeling language, but they also make it much more challenging to create "good schemas." The question whether XML Schema is an appropriate language as a modeling language (in contrast to a pure schema language, which is mainly intended for defining document classes for the purpose of validation), or whether modeling as a more abstract way of defining documents should be left to other approaches is an unresolved question, but so far there is no established modeling language which could provide higher level modeling features than XML Schema [48, 61].

3.5.2. Combining XML Schemas

In simple scenarios, it is often the case that a document class is defined as a standalone schema, with no dependencies from other schemas. In more complex cases, however, it is often necessary to combine schemas, so that the schema information required for processing a document is distributed over more than one schema. Two different forms of this configuration are possible:

- *Creating a Homogeneous Vocabulary:* In this case, a schema designer might reuse components from other schemas (such as types or elements/attributes), but the resulting schema does not expose this kind of modular design, and the documents look like document adhering to a simple stand-alone schema. This kind of schema combination often is implemented using *Chameleon Schemas*, which are namespace-less schemas which are reused, and the components of these schemas then take on the namespace of the schema using them.

  The advantage of this design is that the resulting schema looks homogeneous to the outside, even though it has been constructed modularly. The disadvantage of this design is the fact that it hides the commonalities of the reused components,

because they appear under different namespace names, when they appear in more than one schema. This makes it harder to write software that processes these components consistently.

- *Creating a Heterogeneous Vocabulary:* The other approach is to reuse components from schemas with their own namespaces. In this case, the reused components retain their namespace names, and thus can be identified as being reused from a different namespace. If the reused components are elements or attributes, this kind of schema design is visible in documents, because they have to use different namespaces. The advantage of this method is the fact that the reuse is visible when processing documents.
  A popular example for this kind of schema design is HTML, which in it latest version as the Extensible Hypertext Markup Language (XHTML) [45] is an XML document class. XHTML elements must have the namespace name `http://www.w3.org/1999/xhtml` associated with them, which makes them identifiable as XHTML elements. When other vocabularies are embedded within XHTML, they can simply use a different namespace, and processing stages can simply base their processing on the namespace, rather than having to do more complex and error-prone matching of the element names.

Generally, schemas which use different namespaces appear frequently, and it also is a frequent requirement to be able to base the processing of XML documents on the namespaces they are using. The *Namespace-based Validation Dispatching Language (NVDL)* [27] is a language for achieving this. NVDL is more general in the sense that it does not make any assumption about the schemas being used for different namespaces, but the general idea that document processing should be based on namespaces is applicable to combined XML Schemas as well. However, if only validation is required, XML Schema handles the validation of documents being based on multiple schemas internally, so that for this simple case (validation only) no additional tools are required.

3.6. Other Schema Language Approaches

Currently, the dominating XML schema languages are DTDs and XML Schema, the former because it was the first XML schema language and is universally supported, and the latter because it is the official W3C schema language and has been integrated into several key standards of the XML landscape.

Many critics of XML Schema say that it is too complex and that it is hard to handle and understand for non-experts. As a result of some of these criticisms, the RELAX NG [14] has been created, which is much smaller and easier to understand than XML Schema. Since RELAX NG has been defined by very few people, its design is more radical, and it has been designed to only define dependencies between the schema and the document, so features such as DTDs ID/IDREF or XML Schema's identity constraints (which define dependencies within a document), or XML Schemas structural types (which define dependencies with the schema) are not included. Apart from these differences, RELAX NG is not completely different, it also is a grammar-based language, so defining a RELAX NG schema also means defining rules for how elements and attributes can be combined to yield a valid document.

Schematron [28] is a different type of schema language, because it is not grammar-based. It is based on specifying rules, which are checked when validating a document. These rules can check only isolated aspects of a document, or they can have the same effect as grammar-based schemas, if they are specified in a way so that they check the structural composition of all elements. In most cases, Schematron schemas are useful complements to a grammar-based schema, because Schematron makes it easy to check constraints which are impossible to check in grammar-based languages, such as a rule specifying that two attributes of an element must meet a certain condition (for example, the `end` attribute must have a value greater than the `start` attribute).

While rule-based checking as provided by Schematron can also be implemented in regular programming languages, using XML APIs, using Schematron even for relatively simple rule-based validation tasks has the same advantages as using a schema language in general, which are platform independence, easier maintenance, better readability, and the complete separation of validation and other processing stages.

The *Document Schema Definition Languages (DSDL)* [29] framework is not a schema language, but a framework for using different schema languages, and the NVDL, RELAX NG, and Schematron languages mentioned so far are all part of the DSDL framework (there are even more languages, but some of them are not yet fully specified at the time of writing). The basic idea of DSDL is that validation is not handled best by one very complex schema language, but that different validation tasks are best handled by separate languages, and that it should be easy for user to build a validation pipeline for validating documents against more than one schema. DSDL can be viewed as the component-inspired alternative to the monolithic XML Schema approach.

As yet another example for other schema languages, the *Character Repertoire Validation for XML (CRVX)* [59] language is an example that even seemingly simple tasks may be handled best by a schema language. CRVX can be used to make sure that XML documents do not contain characters from unwanted character repertoires. For example, if a database does not support the full Unicode character repertoire, it does not make sense to accept documents containing unsupported characters. CRVX can be used to filter documents based on character repertoire issues, and is thus useful in combination with legacy software or other software having character repertoire limitations.

3.7. Compound Documents

While documents in many cases contain all information (and thus can be considered as being standalone documents), it is also often useful to have some mechanism to use compound documents, which have dependencies with other documents, which have to be resolved at different stages of a processing pipeline. It is important to keep in mind that any kind of compound document works on a specific level of document interpretation, some may be low-level mechanisms required for even the most basic processing of a document, while others might me more abstract and are only required when processing reaches the respective abstraction layer.

One of the first approaches for specifying a format for packaging multiple document fragments was the *SGML Document Interchange Format (SDIF)* [23], but it never was widely implemented. SDIF was an approach to solve one problem of SGML, the handling

of external entities. External entities (described in more detail in Section 3.7.1 allow documents to contain references to external parts, and when processing the document, the parser resolves the entity reference and replaces it with the entity itself. The problem is how to specify the reference to the external entity, and how to ensure that the reference remains valid, even if the document is processed in different environment.
As mentioned above, mechanisms for compound documents can work on different levels, and it is sometimes hard to decide which level is most appropriate. For the mechanisms described in the following sections, Lease [37] describes a compromise proposal which tries to balance the strengths and weaknesses of all these mechanisms.

3.7.1. External Entities

External entities are XML's include mechanism, they can be used to include external parts of a document. Even though the mechanism basically is an include, it works a bit differently, as shown in the example in Figure 12. The most important difference to a plain inclusion mechanism is that external entities first have to be declared (in the DTD or the internal subset), which creates a mapping between an entity name and a system identifier. If the entity shall be used, special markup is used for specifying the point of the inclusion. The parser then replaces the entity reference with the content of the external entity, and continues parsing with the included content (which is why external entities can be nested).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" [
     <!ENTITY head  SYSTEM "head.xml">
     <!ENTITY header SYSTEM "header.xml">
     <!ENTITY footer SYSTEM "footer.xml">
]>
<html xmlns="http://www.w3.org/1999/xhtml">
      &head;
      <body>
           &head;
              <h1>External Entities in an XML Document</h1>
              <p>This is the actual document content...</p>
          &footer;
      </body>
</html>
```

Figure 12: External Entities in an XML Document

While external entities provide a mechanism for inclusion in XML which is supported by all XML implementations (because it is part of the XML specification), they suffer from the same problem as SGML external entities, which is the question of how collections of documents and document fragments can be managed and packaged (this is the problem that

SDIF wanted to solve for SGML). For XML, the mechanism of *XML Catalogs* [55] has been designed, which defines an entity catalog that maps both external identifiers and arbitrary URI references to URI references. This way, it is possible to exchange and move collections of documents and document fragments without having to update the entity identifiers manually.
While external entities (in particular when used with XML catalogs) provide a convenient way to implement compound documents, this technique also has some disadvantages. The most important disadvantage is that inclusion is part of the parsing process, which means that it is principally impossible to parse and further process fragments without resolving the external entities. Sometimes it may be preferable to parse and process documents and document fragments without resolving inclusions, and if this is required, the newer technology of *XML Inclusions (XInclude)* is better suited than external entities.

3.7.2. XML Inclusions (XInclude)

External entities are part of XML itself, so they are recognize and processed by an XML parsers. This is a very low-level mechanism and may sometimes conflict with requirements where documents should be parsed and processed without resolving references to external resources. This is possible with the *XML Inclusions (XInclude)* [41], which is a separate recommendation and defines inclusions on the Infoset level (as opposed to the markup level). XInclude works by specifying inclusion statements using XML Namespaces, in this case the XInclude namespace. By recognizing and processing elements from this namespace, inclusions are resolved.
The advantage of this approach is the decoupling of the parsing process and the resolution of inclusions, which means that documents can be parsed and processed without resolving external references. Basically, a parser will read a document containing XInclude instructions, and represent these as element nodes using the XInclude namespace, for example in an Infoset-based representation. If required, the Infoset can then be processed by recursively processing the tree, replacing XInclude elements with the included fragments, and continuing this process until all XInclude elements have been processed.
Since XInclude is a separate recommendation and not part of XML itself, it has to be supported by XML processing software, otherwise XInclude elements will not be recognized and cannot be resolved. XInclude support is increasingly finding its way into XML parsers and other XML software, and because it is rather easy to implement, it is very likely that XInclude support will become a standard XML technology which is widely supported.

3.7.3. XML Linking Language (XLink)

Compound documents are a way to use documents which are composed out of several fragments. For the final recipient of the document, it is probably transparent whether the document has been assembled from several fragments, or has been one fragment from the very beginning. In the world of hypertext, however, inclusion (or *transclusion*, where the inclusion remains visible in the resulting document) is only one special case. Hypertext also includes references between documents which should not be used for inclusion, but for

providing navigational aids, so that users can follow these references when encountering them. Web links are the most common example for this, they are links to other documents, but they are presented as navigational aids, rather than replacing them with the content of the document they are referencing.
While Web links are built into XHTML, there is no built-in linking mechanism in XML. This role is played by the *XML Linking Language (XLink)* [16], which defines how to embed linking information into XML documents. XLink's linking model is far more powerful than HTML, allowing out-of-line links, typed links, and many-to-many links. These linking features of XLink, however, are not so important for compound documents, and are explained in detail by Wilde [56].
For compound documents, XLink may be interesting because it supports embedding of content, which is similar to inclusion as discussed in the previous sections about external entities and XInclude. XInclude's embedding of content, however, is more inspired by user-interface issues, where the processing of embedding XLinks should cause the embedded content to be presented in the context of the originating XLink. Technically, it is not specified where XLinks have to be processed, so this can either be done by or close to user interfaces, or in earlier components of the processing chain, for example in proxies mapping XLinks to other technologies [8]. In such a setting, embedding XLinks can be processed by inserting the referenced content (maybe augmenting it with information about its origin), while navigational XLinks can be substituted with linking mechanisms of the target format (for example simple Web links for XHTML).

References

[1] Adobe Systems Inc. PDF Reference: Version 1.3. Addison Wesley, Reading, Massachusetts, 2nd edition, January 2000.

[2] Ron Ausbrooks, Stephen Buswell, David Carlisle, St´ephane Dalmas, Stan De-vitt, Angel Diaz, Max Froumentin, Roger Hunter, Patrick Ion, Michael Kohlhase, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) Version 2.0 (2nd Edition). World Wide Web Consortium, Recommendation REC-MathML2-20031021, October 2003.

[3] Dave Beckett. RDF/XML Syntax Specification (Revised). World Wide Web Consortium, Recommendation REC-rdf-syntax-grammar-20040210, February 2004.

[4] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. Internet proposed standard RFC 3986, January 2005.

[5] Tim Berners-Lee, James A. Hendler, and Ora Lassila. The Semantic Web. Scientific American, 284(5):34–43, May 2001.

[6] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. World Wide Web Consortium, Recommendation REC-xmlschema-2-20041028, October 2004.

[7] Scott Boag, Donald D. Chamberlin, Mary F. Ferna´ndez, Daniela Florescu, Jonathan Robie, and Jérôme Simeón. XQuery 1.0: An XML Query Language. World Wide Web Consortium, Candidate Recommendation CR-xquery-20051103, November 2005.

[8] Niels Olof Bouvin, Polle T. Zellweger, Kaj Grønbæk, and Jock D. Mackinlay. Fluid Annotations Through Open Hypermedia: Using and Extending Emerging Web Standards. In Proceedings of the Eleventh International World Wide Web Conference, pages 160–171, Honolulu, Hawaii, May 2002. ACM Press.

[9] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, Franc¸ois Yergeau, and John Cowan. Extensible Markup Language (XML) 1.1. World Wide Web Consortium, Recommendation REC-xml11-20040204, February 2004.

[10] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and Franc¸ois Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). World Wide Web Consortium, Recommendation REC-xml-20040204, February 2004.

[11] James Clark and Steven J. DeRose. XML Path Language (XPath) Version 1.0. World Wide Web Consortium, Recommendation REC-xpath-19991116, November 1999.

[12] James Clark. Comparison of SGML and XML. World Wide Web Consortium, Note NOTE-sgml-xml-97121, December 1997.

[13] James Clark. XSL Transformations (XSLT) Version 1.0. World Wide Web Consortium, Recommendation REC-xslt-19991116, November 1999.

[14] James Clark. RELAX NG Specification. Organization for the Advancement of Structured Information Standards, Committee Specification, December 2001.

[15] John Cowan and Richard Tobin. XML Information Set (Second Edition). World Wide Web Consortium, Recommendation REC-xml-infoset-20040204, February 2004.

[16] Steven J. DeRose, Eve Maler, and David Orchard. XML Linking Language (XLink) Version 1.0. World Wide Web Consortium, Recommendation REC-xlink-20010627, June 2001.

[17] Patrick Durusau and Matthew Brook O'Donnell. Just-In-Time-Trees (JITTs): Next Step in the Evolution of Markup? In Proceedings of 2002 Extreme Markup Languages Conference [46].

[18] Mary F. Ferna´ndez, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). World Wide Web Consortium, Candidate Recommendation CR-xpath-datamodel-20051103, November 2005.

[19] Jon Ferraiolo, Jun Fujisawa, and Dean Jackson. Scalable Vector Graphics (SVG) 1.1 Specification. World Wide Web Consortium, Recommendation REC-SVG11-20030114, January 2003.

[20] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. Internet proposed standard RFC 2616, June 1999.

[21] Oliver Goldman and Dmitry Lenkov. XML Binary Characterization. World Wide Web Consortium, Note NOTE-xbc-characterization-20050331, March 2005.

[22] International Organization for Standardization. Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML). ISO 8879, 1986.

[23] International Organization for Standardization. Information Processing — SGML Support Facilities — SGML Document Interchange Format (SDIF). ISO 9069, 1988.

[24] International Organization for Standardization. Information and Documentation — Electronic Manuscript Preparation and Markup. ISO 12083, 1994.

[25] International Organization for Standardization. Information Technology — Document Description and Processing Languages — HyperText Markup Language (HTML). ISO/IEC 15445, May 2000.

[26] International Organization for Standardization. Information Technology — EC-MAScript Language Specification. ISO/IEC 16262, June 2002.

[27] International Organization for Standardization. Information Technology — Document Schema Definition Languages (DSDL) — Part 4: Namespace-based Validation Dispatching Language — NVDL. ISO/IEC 19757-4, May 2005.

[28] International Organization for Standardization. Information Technology — Document Schema Definition Languages (DSDL) — Part 3: Rule-based Validation — Schematron. ISO/IEC 19757-3, June 2005.

[29] International Organization for Standardization. Information Technology — Document Schema Definition Languages (DSDL) — Part 1: Overview. ISO/IEC 19757-1, February 2005.

[30] International Press Telecommunications Council. NewsML Version 1.2 — Functional Specification, October 2003.

[31] Ian Jacobs and Norman Walsh. Architecture of the World Wide Web, Volume One. World Wide Web Consortium, Recommendation REC-webarch-20041215, December 2004.

[32] Michael Kay. XSL Transformations (XSLT) Version 2.0. World Wide Web Consortium, Candidate Recommendation CR-xslt20-20051103, November 2005.

[33] Pekka Kilpela¨inen. SGML & XML Content Models. Technical Report Department of Computer Science Report C-1998-12, University of Helsinki, Helsinki, Finland, May 1998.

[34] Donald Ervin Knuth. The TEXbook. Addison Wesley, Reading, Massachusetts, 1984.

[35] Leslie Lamport. LaTeX: A Document Preparation System. Addison Wesley, Reading, Massachusetts, 2nd edition, August 1994.

[36] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. World Wide Web Consortium, Recommendation REC-rdf-syntax-19990222, February 1999.

[37] Karen Lease. External Entities and Alternatives. In Proceedings of XML Europe 2000, Paris, France, June 2000.

[38] Arnaud Le Hors, Philippe Le H´egaret, Lauren Wood, Gavin Thomas Nicol, Jonathan Robie, Mike Champion, and Steven Byrne. Document Object Model (DOM) Level 3 Core Specification. World Wide Web Consortium, Recommendation REC-DOM-Level-3-Core-20040407, April 2004.

[39] Eve Maler and Jeanne El Andaloussi. Developing SGML DTDs: From Text to Model to Markup. Prentice-Hall, December 1995.

[40] Murali Mani. EReX: A Conceptual Model for XML. In Zohra Bellahsene, Tova Milo, Michael Rys, Dan Suciu, and Rainer Unland, editors, Proceedings of Second International XML Database Symposium, volume 3186 of Lecture Notes in Computer Science, pages 128–142, Toronto, Canada, August 2004. Springer-Verlag.

[41] Jonathan Marsh and David Orchard. XML Inclusions (XInclude) Version 1.0. World Wide Web Consortium, Recommendation REC-xinclude-20041220, December 2004.

[42] Eric A. Meyer and Bert Bos. CSS3 Introduction. World Wide Web Consortium, Working Draft WD-css3-roadmap-20010523, May 2001.

[43] Nilo Mitra. SOAP Version 1.2 Part 0: Primer. World Wide Web Consortium, Recommendation REC-soap12-part0-20030624, June 2003.

[44] Theodor Holm Nelson. Embedded Markup Considered Harmful. World Wide Web Journal, 2(4):129–134, 1997.

[45] Steven Pemberton. XHTML 1.0: The Extensible HyperText Markup Language (Second Edition). World Wide Web Consortium, Recommendation REC-xhtml1-20020801, August 2002.

[46] Proceedings of 2002 Extreme Markup Languages Conference, Montr´eal, Canada, August 2002.

[47] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 Specification. World Wide Web Consortium, Recommendation REC-html401-19991224, December 1999.

[48] Arijit Sengupta and Erik Wilde. The Case for Conceptual Modeling for XML. Technical Report TIK Report No. 244, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zu¨rich, Switzerland, February 2006.

[49] C. Michael Sperberg-McQueen and Lou Burnard. The Text Encoding Initiative Guidelines (P4). Technical report, Text Encoding Initiative, Oxford — Providence — Charlottesville — Bergen, March 2002.

[50] Jeni Tennison and Wendell Piez. The Layered Markup and Annotation Language. In Proceedings of 2002 Extreme Markup Languages Conference [46].

[51] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures Second Edition. World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028, October 2004.

[52] Eric van der Vlist. XML Schema. O'Reilly & Associates, Sebastopol, California, June 2002.

[53] Fabio Vitali, Angelo Di Iorio, and Daniele Gubellini. Design Patterns for Descriptive Document Substructures. In Proceedings of 2005 Extreme Markup Languages Conference, Montr´eal, Canada, August 2005.

[54] Norman Walsh and Leonard Muellner. DocBook: The Definitive Guide. O'Reilly & Associates, Sebastopol, California, July 1999.

[55] Norman Walsh. XML Catalogs. Organization for the Advancement of Structured Information Standards, Committee Specification 1.0, October 2002.

[56] Erik Wilde and David Lowe. XPath, XLink, XPointer, and XML: A Practical Guide to Web Hyperlinking and Transclusion. Addison Wesley, Reading, Massachusetts, July 2002.

[57] Erik Wilde and Kilian Stillhard. Making XML Schema Easier to Read and Write. In Poster Proceedings of the Twelfth International World Wide Web Conference, Budapest, Hungary, May 2003.

[58] Erik Wilde. XML Technologies Dissected. IEEE Internet Computing, 7(5):74–78, September 2003.

[59] Erik Wilde. Metaschema Layering for XML. In Robert Tolksdorf and Rainer Eckstein, editors, Proceedings of Berliner XML Tage 2004, pages 106–120, Berlin, Germany, October 2004.

[60] Erik Wilde. Semantically Extensible Schemas for Web Service Evolution. In Liang-Jie Zhang and Mario Jeckle, editors, Web Services — Proceedings of the 2004 European Conference on Web Services, volume 3250 of Lecture Notes in Computer Science, pages 30–45, Erfurt, Germany, September 2004. Springer-Verlag.

[61] Erik Wilde. Towards Conceptual Modeling for XML. In Rainer Eckstein and Robert Tolksdorf, editors, Proceedings of Berliner XML Tage 2005, pages 213–224, Berlin, Germany, September 2005.