Structured Templates for Authoring Semantically Rich Documents

Vincent Quint, Irène Vatton

Structured Templates for Authoring Semantically Rich Documents

Vincent Quint
INRIA Rhône-Alpes
655 avenue de l'Europe
38334 Saint Ismier, France
vincent.quint@inria.fr

Irène Vatton
INRIA Rhône-Alpes
655 avenue de l'Europe
38334 Saint Ismier, France
irene.vatton@inria.fr

ABSTRACT

Structured documents associate explicit semantics with content, but authoring rigorously structured documents is a very difficult task. We present a new approach to this issue that adds schema-level information to the popular web formats. This makes editing highly structured documents easier, while ensuring that documents are valid. It is also an easy way to publish semantically rich documents on the web. The impact of this approach on authoring tools is discussed and its implementation in the Amaya editor is briefly presented.

Categories and Subject Descriptors

I.7 [**Document and Text Processing**]: Document Preparation—*Languages and systems, Markup languages*

General Terms

Design, Experimentation

Keywords

document models, microformats, semantic XHTML, document authoring, structure editing, document templates

1. INTRODUCTION

Documents carry very rich semantics, but only human readers can really take advantage of it. For a long time researchers have tried to process some of the document semantics with computers. They are basically following two different approaches. One consists in relying only on the content of the document (the text, the pictures it contains) to "understand" its meaning. Another approach is to explicitly encode semantics, in addition to the document content, in the document itself or in some external resources associated with it. In this paper, we take the latter approach, following the vision of the semantic web, as defined by T. Berners-Lee:

"The concept of machine-understandable documents does not imply some magical artificial intelligence which allows machines to comprehend human mumblings. It only indicates a machine's ability to solve a well-defined problem by performing well-defined operations on existing well-defined data. Instead of asking machines to understand people's language, it involves asking people to make the extra effort" [1].

The semantic web offers a collection of models and languages such as XML, XML Schema, RDF, OWL, SPARQL to handle various aspects of document semantics under the form of "well-defined data". Each model or language plays a different role, and all are well organized in the so-called semantic web layer cake [4]. But the "extra effort" that people have to make to produce these data is the key issue, that probably explains why the semantic web is not deploying as fast as the traditional web. Most document authors do not spend additional time to formally encode in a computer language (some aspects of) the information and thoughts they have already expressed in natural language or through drawings and pictures.

We address this issue in this paper, focusing on the first levels of the semantic web, namely XML and its schema languages, as well as the associated technologies. We present an approach that helps authors to safely produce "well-defined data" while they are writing documents.

XML's goal is to explicitly represent the logical structure of documents and data. With this structure, programs can securely and unambiguously identify the various parts of a document or a data base, and they get some hints about the type of information located in each part. Structure provides context for the content. This structural context is exploited by programs, which can then perform powerful operations.

XML is now widely spread. It is used to represent many different types of document structures. It encodes structured text (XHTML, DocBook, TEI, etc.), graphics (SVG, X3D), mathematics (MathML), temporal structures in multimedia presentations (SMIL), document layout (XSL-FO), sophisticated forms (XForm), and more. In each case, the explicit semantics represented by the XML structure is different, but it always allows a particular class of programs to offer useful services.

Unfortunately, the issue raised above about the difficulty of getting the appropriate structure well encoded by human producers still makes the deployment of this technology very

SADPI-07, May 21-22, 2007, Montpellier, France.

slow. The well-known case of HTML is a good example of this difficulty. Although HTML was created before XML, it is based on the same concepts. HTML is defined as an SGML application, i.e. as a structured document language. But web authors, and the tools they use as well, tend to ignore that fact. Most of the time, they just try to produce pages that are good enough for their favorite web browser, not paying any attention to other tools that could possibly perform other kinds of treatments. As a result, many applications have simply to ignore the markup supposed to encode the document structure and they can only try to make sense from the content.

To avoid this, better authoring methods are needed. New methods and tools should alleviate the author's task of encoding the details of the document structure. But most XML vocabularies are semantically richer than HTML, and more complex, which makes the issue more difficult. A significant difference is rendering. In HTML, the structure is tightly bound to its presentation. In the absence of any stylistic information, a web browser is able to display a HTML document in the way it is intended by its author. In XML, this is not usually the case. XML's approach is to make presentation independent from structure and to allow the same structure to be rendered for human consumption in very different ways.

There are actually several means to generate a concrete representation from an abstract XML structure. XSL is one of them. This presentation language was specifically designed for XML structures. It is quite powerful, but it also implies complex processing. It works in two steps. First, the original, abstract, XML structure is transformed into another XML structure (XSL-FO) which represents presentational semantics. In the second step, this structure is used to generate the graphical representation of a document, often in the PDF format. This process implies that authors have to create the abstract structure of their document without immediate visual feedback. Some attempts have been made [10] to embed this transformation-based formatting process in interactive authoring tools, but this makes tools very complex.

Another method for publishing XML documents, which is very commonly used, is based on the ubiquitous support for HTML and its companion style language CSS (cascading style sheets) [7]. The principle consists in generating automatically a HTML+CSS representation of XML documents, thus allowing them to be viewed on the very broad range of devices available on the web. XSL is also used in this framework, but only the first step of its process is involved. The transformation step generates directly a HTML document and associates predefined CSS style sheets with it. With this method, the document is first authored in some XML language, following a schema well suited to the document type. The XML document is then transformed into HTML for publication. This method is still complex, as the authoring phase is performed directly on the XML structure, and requires authors to be comfortable with the language and its tools. Also, when the document is published on the web, most of the semantics carried by the original XML structure is lost, and applications that process the HTML output cannot do much with it, except displaying or printing it. It should be noted however that using the full XSL process to produce PDF files does not make any significant difference in that regard.

In this paper, we are exploring a different way to authoring and publishing semantically rich structured documents that can be used everywhere, with the most common tools for accessing electronic documents: web browsers. The paper is structured as follows: the next section presents the approach based on the combination of semantic XHTML and schema-like information; section 3 explains how this approach helps an authoring tool, and section 4 discusses the benefits of the approach.

2. A NEW APPROACH

As an alternative to the complex process described above, we are experimenting with a new approach that is simpler while keeping the advantages of a rigorous well-defined structure, a universal publication format, and semantic information delivered to the end user.

To achieve this goal, the main idea is to put together, in a single information resource, the various structures and languages involved in the traditional production process described in the previous section. This allows a single tool to perform efficiently and consistently all the operations needed. This allows also authors to work freely, with complete support for all the tasks they have to perform, at any time.

More precisely, the different levels of representation that have to be put in the information resource are the following:

- HTML encoding (or, better, XHTML), for display during the authoring process and for publication;
- Style information, to specify how to visually render the (X)HTML representation during the authoring phase, and for publication in different contexts;
- XML structure, to represent the details of the logical structure of the document, thus carrying some valuable semantics;
- Schema information, to define and constrain the XML structure;
- Mapping information, to establish the correspondence between the XML structure (as defined by schema information) and the (X)HTML encoding.

2.1 XHTML and style

In the production process, XHTML plays two roles. It is used to format and display the document during the editing phase. It is also used as the final publishing format. Using the same format for both roles allows authors to work more comfortably, as they can see at any time what will really be delivered when the document is finished. The term WYSIWYG is not really appropriate here, as "What You Get" is not completely defined with XHTML: the exact form of the document depends on the browser used to display the document, the preferences set by the user, and the style sheets associated at that time. The style of interface is actually closer to direct manipulation [9], as the author really manipulates the final representation that will be produced in the end: the XHTML representation.

XHTML was preferred to HTML for several reasons. The first reason is that XHTML is an XML language and it is much easier to use it in combination with other XML languages. Also, with XHTML 1.1, we have a clean separation

between structure and style, which offers more flexibility for using style sheets and thus adapting documents to the environment where they are used.

For specifying style, we use CSS [7]. Like XHTML, it is supported by all web browsers, and it allows the presentation of documents to be changed freely, depending on the environment or on the task to be performed. For instance, CSS offers advanced features such as the `@media` mechanism that allows a style sheet to specify different styles for different kinds of devices (printer, screen, projector, handheld, TV, braille, speech, etc.).

2.2 XML structure

Although XHTML is an XML language, it does not carry much semantics. Its elements have a rather general purpose, like division (`div`), paragraph (`p`), heading (`h1`), etc. They allow many different types of documents to be represented, but at a very low level: in a web page, a `div` may be used to group news items, while the same element may represent a chapter in technical documentation. Without additional information attached to it, a plain division does not tell much about its content.

To solve this issue, we follow the principle of semantic XHTML and microformats [5]. It consists in defining a rich structure (a technical documentation, the organization of a web page, the contact information of a person) in terms of another, less specialized language (typically XHTML), by stating guidelines and conventions for using the lower level language.

This approach has many advantages. By using a popular web markup language without any extension, documents can be accessed with any web browser. The structure constrained by the rules of the microformat provides detailed information that can be exploited by CSS style sheets to fine tune the style and the layout of documents on different devices. All the details of the structure are available when the document is delivered over the web. Applications of different kinds can then extract and use information from these web pages and provide valuable services to end users.

With this approach, we consider XHTML as a (almost) semantically neutral format for constructing arbitrary tree structures. Element `span` can be used for the smallest pieces of information, `p` for assembling `span`s, and `div` for building hierarchies on top of `p`s. Some other XHTML elements, like lists or tables, may also be used for representing sequential or tabular structures. Semantics is added through a few XHTML attributes, such as `class` or `title`, that can be associated with almost every element. These attributes are used to carry the information that is lacking from the element name. As an example, consider the following XML structure representing the contact information for a person (for instance an author of this article):

```
<person>
  <full-name>Irène Vatton</full-name>
  <address>INRIA Rhône-Alpes</address>
  <email>irene.vatton@inria.fr</email>
</person>
```

Example 1: an XML structure

With semantic XHTML, the same information could be represented by:

```
<p class="vcard">
  <span class="fn">Irène Vatton</span><br/>
  <span class="adr">INRIA Rhône-Alpes</span><br/>
  <span class="email">irene.vatton@inria.fr</span>
</p>
```

Example 2: a semantic XHTML structure

From the semantic point of view, there is not much difference between both pieces of code. The tree structure is almost the same. Element names are different, but the information carried by element names in the XML version is encoded in the `class` attributes in the XHTML version. A few more elements (`br`) are present in the XHTML version, but this is not a problem: any processor can just ignore them if they are considered useless. Both representations enable the same kind of processing. An XSLT engine for instance can perform valuable transformations on both pieces of code. XQuery queries can extract as much information from both versions as well. An address book application can recognize the various pieces of information and process them safely.

The big difference however is that the XHTML version can be sent immediately to a web browser, and even without any style sheet, something sensible will be displayed. If a CSS style sheet is associated with the document, it could display the information nicely, taking advantage of the semantics carried by the attributes.

In Example 2, the values of the `class` attribute have been carefully chosen. They are taken from the hCard microformat. If this piece of code is received, as part of a web document, by a browser that knows about hCard (thanks to a plugin for instance), this browser could update the personal address book of its user with the contact information of this author. More generally, using attribute values that are part of well-known microformats allows documents to be processed by various applications that are already deployed on the web.

2.3 Schema information

The strength of XML is not only the structure it brings to documents, but also the schemas that control this structure. Schemas serve different purposes. First, they specify document types, by defining the elements and attributes that can be used as well as the rules for combining them into a tree structure. Based on this specification of a document type, schemas are used for validation, i.e. for checking if a given document follows the definitions and rules of its associated schema. Programs that create or generate documents also use schemas as drivers, to make sure they produce valid documents.

Validation is very important. It is because the structure of a document is rigorously specified by a schema, and can be checked by validation, that it is possible to securely associate some semantics with it. Valid documents make processing simpler. There is no need to consider other cases than those explicitly mentioned by the schema. Valid documents also make processing more reliable. There is no unexpected cases that could raise errors in a program or that could lead to unpredictable behaviours.

Several schema languages are available for XML, ranging from DTD to RELAX NG and XML Schema. All schemas are external resources that are referred from documents. For an authoring tool, this has some consequences. In particular,

an authoring tool has to know what elements are allowed at each position in the document structure, to guide the author and to check that new elements created or pasted in the document are acceptable. With an external schema, this is not a simple task. It requires that the tool be able to match a particular position in the structure with the corresponding rules in the schema. A solution to this issue is to explicitly list all possible elements at every position in a document structure. While this may look costly, it is actually not a problem in the context of semantic XHTML.

Defining the structure of a new document type with a schema is a complex task, which requires that all elements be defined from scratch. Even the most common elements have to be defined, while they already exist in many other schemas. Reusing an existing schema in a new schema could address this concern, but at the moment the proposed solutions are complex. Our approach is to build on semantic XHTML again: as the most common text structures are readily available in XHTML, there is no need to redefine them. We can just use them wherever they fit in the document type. Paragraphs, itemized lists, emphasized character strings, tables, etc. can just be borrowed from XHTML. This makes the document type definition much simpler. Only the structures that are specific to a particular document type need to be defined. This simplification greatly reduces the cost of including some schema-level information in the document itself, as proposed above.

To implement this approach, we have designed a language that plays a role similar to a schema language, but for semantic XHTML documents: XTiger (Extensible Templates for Interactive Guided Editing of Resources). This language has two main features: 1) it can define semantic XHTML structures, such as the one of Example 2; 2) these building blocks can then be used at some well identified locations in the document structure. The document structure itself, at the top level, is the usual structure of a XHTML document, which is called a template. The template contains both XHTML elements, that constitute the skeleton of all documents of a given type, and some XTiger elements. Although all examples in this paper are based on the XHTML language, it should be noted that XTiger is independent from XHTML. It can be used with any other XML language, even with multiple languages in the case of compound documents.

In a template, all structure definitions are grouped in a unique XTiger element `xt:head` that is inserted in the head of the XHTML document (we use prefix `xt:` in front of all XTiger element names to distinguish the XTiger namespace from the XHTML namespace; XHTML names have no prefix). Two structure definition elements are available, `xt:component` and `xt:union`. Building blocks defined by `xt:component` have a name that allows them to be referred from elsewhere in a template and they contain a piece of semantic XHTML as well as some XTiger elements. Example 3 shows the definition of a component named “author”, which specifies the structure of Example 2. Note the `xt:use` elements. These elements indicate that, at this location in the structure, only character strings (`types="string"`) can be used. The `xt:use` elements also contain an initial value that can be freely replaced by any other value when editing a document. Example 4 shows another component definition, which is used to define the structure of the examples that appear in this article.

```
<xt:component name="author">
  <p class="vcard">
    <span class="fn"><xt:use types="string">Author
                name</xt:use></span><br/>
    <span class="adr"><xt:use types="string">Snail
                mail address</xt:use></span><br/>
    <span class="email"><xt:use types="string">email
                address</xt:use></span>
  </p>
</xt:component>
```

Example 3: a semantic XHTML structure

```
<xt:component name="code-example">
  <div class="example">
    <pre><xt:use types="string">
                Some code</xt:use></pre>
    <p class="caption"><xt:use types="string">
                Caption</xt:use></p>
  </div>
</xt:component>
```

Example 4: defining a component

The other structure definition element available in XTiger is `xt:union`. It defines components that can be chosen among a list of options. For instance, to define the allowed content of a section in this article, we could use a union named ”section-content”. Example 5 states that section-content may be either the component defined in Example 4, or a XHTML element `p` (paragraph), or a XHTML element `ul` (unnumbered list).

```
<xt:union name="section-content"
          include="code-example p ul"/>
```

Example 5: defining a union

With this kind of structure definition, we can describe the whole structure of a type of document in a template. A template is constituted of XHTML elements plus XTiger elements. The XTiger elements indicate where and how structures defined as components and unions, as well as plain XHTML elements, can be used. There are four such XTiger elements. We have already seen the `xt:use` element, which indicates that a single, mandatory element must appear at its location. The `xt:use` element also indicates the allowed type(s) for this element. When a sequence of elements is allowed, the `xt:repeat` element is used. It always contains a `xt:use` element, to specify the type(s) of these repeated elements. Example 6 is an excerpt from the template describing this article. It specifies that the XHTML `div` element that contains information about the authors after the title must hold one to five ”author” components (those defined in Example 3).

```
<div class="authors">
  <xt:repeat minOccurs="1" maxOccurs="5">
    <xt:use types="author"/>
  <xt:repeat>
</div>
```

Example 6: repeating a component

Similarly, there is a `xt:option` element for optional structures. It is equivalent to a `xt:repeat` element with minOccurs=0 and maxOccurs=1. The fourth element is called `xt:bag`. A bag can contain any number of elements, which can themselves contain other elements, but the types of all these elements, at any level within the bag, is constrained by the `types` attribute of the `xt:bag` element. For instance, for this article, we could define a component named "section" as a XHTML `div` that, in addition to the `h2` heading, would contain only examples, paragraphs and lists (see Example 7, which refers to the structure defined in Example 5).

```
<xt:component name="section">
  <div class="section">
    <xt:use types="h2"/>
    <xt:bag types="section-content"/>
  </div>
</xt:component>
```

Example 7: a bag

There are a few more elements in the XTiger language, and the elements presented above have a few more attributes, but this is the core of the language. XTiger is a simple language. For more details about XTiger, see [6].

2.4 Mapping information

In the traditional publication process of XML documents, when targeting the web as a publication medium, XML structures have to be converted into XHTML. This transformation, often performed by a XSLT sheet, is based on a mapping process, where the XHTML structures to be generated are associated with XML structure patterns from the source document.

With XTiger, mapping is useless, as well as transformation. All structures are defined from scratch as XHTML structures by the XTiger language. Mapping becomes immediate, as the semantic structure and the XHTML code coexist in the same file, the one embedding the other.

If a pure XHTML document is needed for publication, the XTiger elements can be removed from the document. This is a very simple operation, but removal of XTiger elements is not necessary. Experience has shown [3] that the XTiger code usually increases file size by an acceptable ratio. Moreover, web browsers are not disturbed by the XTiger elements interspersed in the XHTML code. Browsers are built to ignore elements they do not know. This obviously applies to XTiger. This is a significant advantage, as documents can be distributed on the web without any further processing, as soon as the author is finished.

3. HANDLING DIFFERENT STRUCTURES

As we have seen above, a document contains its representation in semantic XHTML as well as structural constraints expressed by XTiger elements. These two structures play different roles, but they are tightly intermixed. For an authoring tool, this raises several issues related to structure manipulation, validation, presentation and editing.

3.1 Semantically rich structure

To process both structures correctly, they first have to be distinguished. Namespaces in XML [2] have been created for that purpose, and they provide the mechanism needed for XTiger. With namespaces, every element can be associated unambiguously with the language to which it belongs. It should be noted however that when using XTiger, attributes are not mixed: XHTML elements never have XTiger attributes and conversely.

Namespaces work well because both XTiger and XHTML structures are hierarchical and share the same content. Elements from one structure are never split into several elements in the other structure. We have carefully avoided complex mixing schemes.

Under this condition, integrating both structures is not a problem, but an advantage, as it greatly helps an authoring tool. When processing an XHTML element in a document, the constraints that apply to it are readily available. By checking the ancestor XTiger elements of the element of interest, a document processor may know what is allowed and what is not. Obviously this applies only to the constraints expressed by the XTiger language. Constraints from the XHTML language still have to be checked in the corresponding DTD. For doing this, only the XHTML elements present in the structure have to be taken into account. XTiger elements must be ignored.

XTiger enables an interesting approach to validity (in the XML sense). Instead of post-hoc validation, it allows an authoring tool to continuously, dynamically ensure that the document it builds is always valid. With that approach, the tool does not need to validate the full document at any time, but it enforces validity for each action it performs, by allowing the author to trigger only operations that keep the document valid. For the XTiger structure it is quite easy. For XHTML the advantages of this approach have already been demonstrated [8].

To make sure the XTiger structure is present wherever it is needed in a document, an authoring tool has to generate it every time new elements are introduced in the document. New elements can be created only within the XTiger elements `xt:use`, `xt:repeat`, `xt:option` and `xt:bag`. They all allow both XHTML elements and XTiger components to be created. When creating a XTiger component, the whole structure of the component is copied from its definition given by a `xt:component` element. In addition, if a `xt:repeat` element is part of the component, additional children may be generated to comply with the minimum number of occurrences required.

When an authoring tool creates a XHTML element that is allowed both by its DTD and by the XTiger structure, the presence of the XTiger structure impacts the way these elements are inserted in the structure. Elements that should be children of a given elements according to the XHTML DTD often have to be inserted deeper in the structure, because of intervening XTiger elements. This is actually the most difficult aspect of handling both structures concurrently. While XHTML structure checking must simply ignore XTiger elements, structure construction must take them into account. Consider Example 8 that specifies the structure of a bibliography at the end of an article.

```
<ol class="bibliography">
  <xt:repeat>
    <xt:use types="bibitem"/>
  </xt:repeat>
</ol>
```

Example 8: a bibliography

It uses the "bibitem" component defined in Example 9.

```
<xt:component name="bibitem">
  <li class="bibentry">
    <span class="auth">
      <xt:use types="string"/>
    </span>
    ...
  </li>
</xt:component>
```

Example 9: a bibliography entry

When the author wants to add a new item in the bibliography, a new XHTML element `li` with `class="bibentry"` is created, but it is not inserted as a child of the `ol` element (which should be done according the XHTML DTD), but two levels deeper (see Example 10).

```
<ol class="bibliography">
  <xt:repeat>
    <xt:use types="bibitem">
      <li class="bibentry">
         <span class="auth">
           <xt:use types="string"/>
         </span>
         ...
      </li>
    </xt:use>
    <xt:use types="bibitem">
      <li class="bibentry">
         <span class="auth">
           <xt:use types="string"/>
         </span>
         ...
      </li>
    </xt:use>
  </xt:repeat>
</ol>
```

Example 10: adding an entry in the bibliography

3.2 Presentation structure

Template elements not only interfere with the logical structure, but also with the presentation structure. In a direct manipulation system this is important, as the logical structure is not displayed only through its XML source code or its DOM tree, but primarily through a formatted representation. For the formatter too, the presence of XTiger elements has to be considered.

Due to the way both structures are mixed, the CSS box model works well for formatting documents with XTiger elements. CSS was designed [7] to format hierarchical structures by associating nested boxes to the nested elements of an XHTML (or XML) structure. As the XHTML elements and the XTiger elements mixed in a document make a unique hierarchical structure, we can associate a CSS box with each XTiger element, while keeping the usual boxes for XHTML elements. The XTiger boxes are specified in CSS as simply as possible. They do not add extra space or any particular positioning. It is just a rectangle that delineates the XHTML elements they contain. That way, XTiger elements do not disturb the formatting of XHTML elements. An author can edit a document while seeing the usual formatting of the XHTML content. This is important for direct manipulation.

Though they do not change the formatting of the document, XTiger boxes are visible under the form or colored rectangles. A different color is associated with each type of XTiger element (see Figure 1). The author can thus perceive and understand the constraints put by the XTiger elements. In the upper part of Figure 1, the following colors are associated to XTiger elements: blue for `xt:use`, purple for `xt:repeat`, yellow for `xt:option`, and green for `xt:bag`.

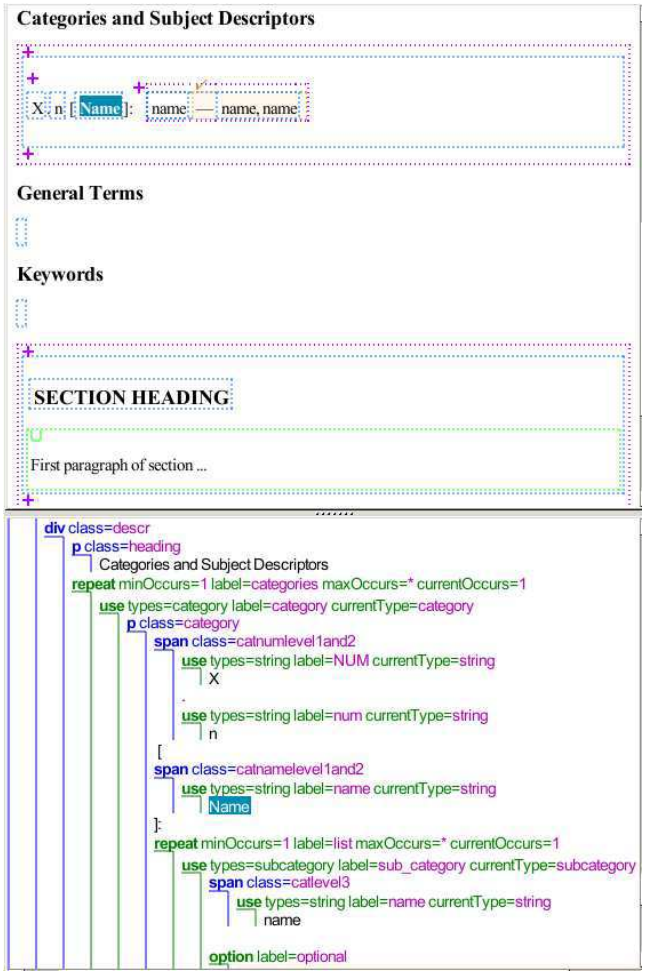

Figure 1: Displaying a XTiger document

3.3 Editing process

The principles discussed above have been implemented and experimented in the Amaya authoring environment [8]. Users can now create new XHTML documents from a XTiger template and edit them following the model described by the template.

When creating a new document, Amaya instantiates the chosen template. It creates a copy of the template which becomes the new document, and it removes the `xt:head` element (i.e. all structure definitions: `xt:component` and `xt:union` elements). It keeps the head separate from the new document instance. To maintain the link between the document instance and its template, it inserts in the new

document an XML Processing Instruction containing the URI of the template, in the same way CSS style sheets are linked from XML documents. The new document is then ready for editing (see Figure 1). Note that editing is allowed only within the colored frames (i.e. XTiger elements). The parts of the template that are not within a XTiger element must stay as they are and can not be edited.

The editing process then follows the principles detailed in the previous section, maintaining both the XHTML structure and the XTiger elements in the same data structure. During the editing phase, the initial document grows and the new elements from both languages are added. Their visual representation is simultaneously reflected on the display, where the user can see both structures. To better understand the structure being built, the user may ask the editor to show the DOM tree in a separate view, which is also updated every time the internal structure is changed. In this view the XTiger elements and XHTML elements are displayed in different colors (lower part of Figure 1).

4. DISCUSSION

XTiger templates may be used to specify the overall structure of a large document, as well as the fine details of some of its parts. This latter feature allows document designers to specify how to use microformats in large documents. In that regard, XTiger is quite different from the templating mechanisms offered by most commons document editors, which only provide the overall structure of a document. Using an XML syntax, template information can be easily integrated in the XHTML structure and carries more information about the document model than templating languages that use only comments or attributes. The template language is very simple (only a few elements and attributes), but experience has shown that it is powerful enough to represent faithfully the structure of many different types of documents. Some templates are available on the web from http://www.w3.org/Amaya/Templates/.

In this paper, XTiger is used only with XHTML. It can actually be used also with other XML document formats. In particular, we have plans to use it with SVG and MathML in compound documents. With these languages, XTiger components could define typical graphical or mathematical structures that occur frequently in some documents. To make that easier, XTiger includes a `xt:library` element that is used to collect component declarations in separate files, called libraries, that can then be shared and reused in several templates. A `xt:import` element allows libraries to be called from the `xt:head` element of a template.

Another important feature of XTiger is that a document can be made available on the web with its XTiger elements without disturbing browsers. All the details of the structure (and the corresponding semantics) of the document are available on the client side. This is not the case when XML documents are transformed into XHTML on the server side. Moreover, keeping the XTiger structure with the published document allows other users to take advantage of it, and to keep working on the document with exactly the same possibilities as the original author. This is very helpful for cooperative work.

We have stressed in the introduction the difficulty of getting explicit semantics properly encoded by authors. With XTiger, the effort of encoding semantics in documents is done when designing templates, not by authors writing documents. Actually, using XTiger not only allows authors to finely structure their documents, but it also saves time for them. When a large piece of structure is represented by a component, inserting an instance of this component in a document requires only a mouse click. If the template is well designed, authors spend most of their time typing in the content, and do not have to worry about structure. This is important, because manipulating structure is not easy for most authors. With XHTML this is indeed complex, as XHTML elements are very general and authors have difficulty to choose the right element for representing the structure they have in mind. XTiger templates provide much appreciated help in that regard. For example, in Figure 1, adding a new Category is done simply by clicking a purple '+' icon and all the required fields are immediately created and displayed.

The key point is then to design templates. This may be compared to designing an XML schema, although building on top of XHTML (or other XML format) reduces the task, as mentioned above. The main issue is to conceive a well thought model and to map it to semantic XHTML. The model must then be encoded as a template. There is currently not much help for this technical task. Although the template designer can use Amaya to build a skeleton in XHTML, she must then edit the source code to introduce XTiger elements. A template design mode in Amaya would certainly ease this task. This is scheduled for future developments.

Developing templates is costly, but this cost must be balanced with the many document instances that will be created from it. It is in fact a valuable investment. Not only document authors save time, but they also produce more valuable documents. Instances built from a template are better structured, carry more explicit semantics, and can be exploited by more applications.

In section 2.2, we have already mentioned address book applications taking advantage of the hCard microformat. Similarly, the bibliography of a scientific article can be encoded in a citation microformat to automatically feed the bibliographic data base of its readers. This kind of benefit applies not only to scientific papers, but to all sorts of documents and applications. In a different area, aggregators such as kritX, iNods or blogcritics collect reviews from weblogs and other web sites and rely on the hReview microformat to find the relevant information. Announcing events on the web with the hCalendar microformat allows users to quickly and safely update their own calendar when they find interesting announcements. Web applications such as Slidy or S5 rely on semantic XHTML to present slides with a number of advanced features. Describing these formats in XTiger templates makes it sure that they are correctly encoded and then that applications can work properly, without asking authors to learn about new formats.

It could be argued that the level of semantics encoded in semantic XHTML is low. This is true, as compared with other layers of the semantic web, for instance. But with tools like Amaya and its XTiger templates, this low-level semantics enables a lot of useful applications. In addition, the deployment of these applications is very smooth, as the formats involved do not require any extension of the existing web infrastructure or tools. Slidy and S5 are good examples: access a Slidy document with your favorite browser and it works immediately. Dowload a few plug-ins and your

browser exploits on the corresponding microformats for you. Put some hReview elements in your blog, and aggregators will reference your comments.

5. CONCLUSION

In this paper, we have presented a new approach to document structure, which is based on the usual concepts of XML documents, structure schemas, and mapping between different levels of representation. But we are using these concepts in a different way, combining them in a single, simple language that offers new possibilities for authoring and publishing semantically rich structured documents.

This approach was implemented in the Amaya authoring environment and is used with various types of documents.

6. ACKNOWLEDGEMENTS

The work presented here was partly funded by the 6th Framework Programme of the European Commission as part of the Palette project (FP6-028038). The implementation of XTiger would not have been possible without the continuous support of W3C for the development of Amaya.

7. REFERENCES

[1] T. Berners-Lee. *What the Semantic Web can represent.* http://www.w3.org/DesignIssues/RDFnot.html, September 1998.

[2] T. Bray, D. Hollander, A. Layman, and R. Tobin. *Namespaces in XML 1.1.* W3C Recommendation, http://www.w3.org/TR/xml-names11/, August 2006.

[3] F. C. Flores, V. Quint, and I. Vatton. Templates, microformats and structured editing. In *Proceedings of the 2006 ACM Symposium on Document Engineering, DocEng 2006*, pages 188–197. ACM Press, October 2006.

[4] J. Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, 16(2):30–37, 2001.

[5] R. Khare. Microformats: the next (small) thing on the semantic web? *IEEE Internet Computing*, 10(1):68–75, 2006.

[6] E. Kia, V. Quint, and I. Vatton. *XTiger Language Specification.* http://www.w3.org/Amaya/Templates/XTiger-spec.html, February 2007.

[7] H. W. Lie. *Cascading Style Sheets.* Phd thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, February 2006.

[8] V. Quint and I. Vatton. Techniques for authoring complex XML documents. In *Proc. 2004 ACM Symposium on Document Engineering*, pages 115–123. ACM Press, October 2004.

[9] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[10] L. Villard and N. Layaïda. An incremental XSLT transformation processor for XML document manipulation. In *WWW2002, the 11th International World Wide Web Conference*, pages 474–485. ACM Press, May 2002.