



Templates, Microformats and Structured Editing

Francesc Campoy Flores
INRIA Rhône-Alpes
655 avenue de l'Europe
38334 Saint Ismier, France
francesc.campoy-
flores@inria.fr

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

Microformats and semantic XHTML add semantics to web pages while taking advantage of the existing (X)HTML infrastructure. This approach enables new applications that can be deployed smoothly on the web. But there is currently no way to describe rigorously this type of markup and authors of web pages have very little help for creating and encoding semantic markup. A language that addresses these issues is presented in this paper. Its role is to specify semantically rich XML languages in terms of other XML languages, such as XHTML. The language is versatile enough to represent templates that can capture the overall structure of large documents as well as the fine details of a microformat. It is supported by an editing tool for producing documents encoded in a semantically rich markup language, still fully compatible with XHTML.

Categories and Subject Descriptors

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

General Terms

Design, Experimentation

Keywords

World Wide Web, document models, microformats, semantic XHTML, document authoring, structure editing, document templates

1. INTRODUCTION

XML was created for exchanging a wide variety of structured documents and data on the web [2]. Although it is possible to send XML over the web and to let clients process that format for presentation, in most cases XML is used on

the server side only, as a source format from which other representations are derived. Documents are transformed into XHTML before delivery to the client. This ensures that information can be presented on many different types of devices, as support for XHTML is ubiquitous nowadays. XML is used upstream in the delivery chain, to model documents and to structure them consistently. The main benefit is that documents are represented on the server with a rich markup language, independently of their presentation, and they can be used in a number of applications.

In this process, the first task is typically to design a document model and to formalize it in a schema or a document type definition (DTD), if such a model is not already available for the type of document to be handled. Before publication or delivery, documents have to be converted into XHTML. This is often achieved by a transformation expressed in a language such as XSLT [5] and a specific transformation sheet has then to be developed.

This process is long and complex. In many cases authors prefer to take another, very different approach and write directly XHTML documents ready for publication. They simplify the process, but they miss the considerable advantages offered by XML.

In this paper we try to combine the advantages of both approaches: a rigorous document structure, but a simple production process. More precisely, the goal is to make it easier for authors to create and edit well structured and semantically rich documents that can be accessed with simple browsers, directly over the web, without requiring complex schemas and transformations, while still allowing documents to provide useful, automatically processable information on both the server and the client.

The next section introduces our approach. It is followed by a presentation of XTiger, a language that implements that approach. Then section 4 shows how XTiger is supported in an authoring environment. Section 5 compares our approach with other similar works and section 6 discusses the results. Finally the conclusion summarizes the main contributions and suggests a few next steps.

2. APPROACH

Consider this article. Its structure, defined by the publisher, has to be followed carefully by the authors. It starts with the title and the list of authors with their address. This is followed by an abstract, some categories, general terms and keywords, with numbers and names extracted from a list of predefined values. Then comes the body of the document as a sequence of sections. The article ends with a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DocEng '06, October 10–13, 2006, Amsterdam, The Netherlands.
Copyright 2006 ACM 1–59593-515-0/06/0010 ...\$5.00.

list of bibliographic entries, which themselves have a well defined structure. Whereas the front matter or the bibliography are rigorously organized, some other parts are not constrained very strongly. A section for instance must start with a heading, but it may contain different types of elements (paragraphs, bulleted lists, figures, tables, examples, etc.) that the author is free to arrange in any suitable way.

On the other hand, to make this paper available on the web, with all the benefits offered by the web (links that readers may click, style suited to the device or to user preferences, etc.), the document should be encoded in XHTML. And to make the production process simple and efficient, authors should be able to create the documents directly under the form used for publication. The issue is that XHTML does not seem to be rich enough to represent all the details of the structure described in the previous paragraph.

Looking closer at the issue, it appears that XHTML can actually do the job, in particular by exploiting the `class` attribute. This attribute gives a more precise role in the structure to elements that are otherwise a bit vague, like `div` (division) or `span`. For instance, the information about the authors can be wrapped in a `div` with an attribute `class="authors"`. In this `div`, the `p` (paragraph) that contains contact information for an author can be assigned an attribute `class="author"`. To refine the structure in this paragraph, the name and the various parts of the address can be separated into different `span` elements, each with a different `class` attribute identifying their role. One can go further and separate the given name from the family name.

This approach is called microformats [10]. It consists in defining a rich structure (an article or the contact information of a person) in terms of another, less specialized language (XHTML), by stating guidelines for using the lower level language. This approach has many advantages:

- Documents can be structured with semantically rich markup.
- No transformation is needed for displaying a document with a web browser: the document is encoded in plain XHTML, without any extension.
- The structure provides detailed information that can be exploited by CSS style sheets [12] 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 extract and reuse information from these web pages.

Let us consider the latter item further. In the structure of an article, we could choose the `class` attributes defined in the hCard microformat [17] to encode author names and addresses. This would allow all hCard-enabled applications on the web to retrieve information about these authors directly from the document. It would be the same for the bibliography of this paper if we use a citation microformat, it would be the same for categories, for general terms, for keywords, etc. if we use the appropriate microformats. This benefit applies to all sorts of documents and applications. An example in a different area is kritX, an aggregator that collects reviews coded in the hReview microformat from weblogs and web sites.

Microformats, also called semantic XHTML, have a number of advantages, but also a few drawbacks. First, more markup is required than for plain XHTML code. Producing markup by hand is tedious and error-prone. Second, these formats are not defined by formal specifications. If the additional semantics of microformats are not correctly encoded in the XHTML markup, most of their benefits are lost: style sheets do not work correctly and applications can not retrieve the information they are supposed to process.

To address these issues, we have developed a language and an editing tool. The tool makes editing microformats easier, simpler, safer and more effective. The language, called XTiger (Extensible Templates for Interactive Guided Edition of Resources), allows semantic XHTML and microformats to be clearly described. The editing tool uses descriptions expressed in this language to help authors to produce valid documents, i.e. documents where the additional semantics of the microformats are correctly encoded.

3. THE XTIGER LANGUAGE

The main role of the language is to describe a *generic structure* in terms of another structure representation language called the *target language*. The target language considered above is XHTML, but it might be any other XML language as well. The generic structure to be described may be a microformat, i.e. the structure that organizes a small part of a document and associates some semantics with it. The contact information of a person or the details of a bibliographic citation discussed above are typical examples. But it could be also a larger piece of information, including a whole document, such as this article, or a slide show (with S5 or Slidy). The generic structure is a model from which document instances are derived. All instances derived from the generic structure are supposed to comply with the constraints expressed in the model.

To give a rough idea of an instance and its template, example 1 shows how the hCard microformat is used for representing the instance an author of this article, while example 2 provides the template for an `author`. Details about the XTiger elements (those with a `xt:` prefix) are provided further in this section.

```
<p class="vcard">
  <span class="fn">Irène Vatton</span>
  <br/>
  <span class="adr">INRIA Rhône-Alpes<br/>
    655 avenue de l'Europe<br/>
    38334 Saint Ismier, France
  </span>
  <br/>
  <span class="email">irene.vatton@inria.fr</span>
</p>
```

Example 1: a microformat

```
<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 br">Address line 1<br/>

```

```

                                line 2...</xt:use>
</span>
<br/>
<span class="email">
  <xt:use types="string">email</xt:use>
</span>
</p>
</xt:component>

```

Example 2: definition of type author

The XTiger language has three main features:

1. It allows specific constructs to be defined as *types* (like **author** in example 2).
2. It allows to *constrain* the use of the target language and these types in the structure of an instance.
3. It provides help to *share and reuse* constructs (types).

XTiger is always used in combination with a target language. It describes a generic structure under the form of a template. A template is a skeleton document, expressed in the target language, which contains elements of the XTiger language in various places (see example 2). The role of these XTiger elements is to specify what elements and attributes of the target language must, should or could be present at these places, possibly with some predefined content or value. That is the core of the language, which specifies the structure and the content of documents.

It is complemented with other features that make the language easier to use. For instance, structure fragments can be defined once and used at several places, in one or several templates. This facilitates a modular construction of templates, by sharing reusable pieces of structure.

XML is a natural choice for the syntax of XTiger, as its main role is to describe structures. In addition, using the same syntax for the structure description language and for the target language can simplify implementation. XML also provides the namespaces mechanism [1] for mixing different languages while still making a clear distinction between them. This distinction allows existing web browsers to simply ignore the XTiger elements and display a template as if these elements were not present. In the following, namespaces are used with prefix **xt:** for all names from the XTiger namespace, while names from the target language are not prefixed.

3.1 Types

In XTiger, types are used to specify pieces of structure that may occur at several places in a template or in several templates.

Types are built from *basic types* using *constructors*. In the initial version of XTiger, there are a few basic types: number, string, boolean. In the future this short list is expected to grow and include datatypes like those from XML Schema. The basic types are used to specify the content of some parts of a document. For instance, the three **span** elements that appear in example 3 use the basic type **string** for their content (element **xt:use** is presented in section 3.2).

Basic types are also used to build complex types. Element **xt:component** is a constructor that creates a new type containing different elements, both from the XTiger language and from the target language. In the template for an article, we could define a type **refbook** that would be useful

in the bibliography to refer to a book (see example 3 for a simplified version). We can similarly define other types of bibliographic citations to refer to articles (**refarticle**), technical reports (**refreport**), etc.

```

<xt:component name="refbook">
  <p class="refbook">
    <xt:repeat minOccurs="1">
      <span class="bibauthor">
        <xt:use types="string"/>
      </span>
    </xt:repeat>
    <span class="title">
      <xt:use types="string">Book title</xt:use>,
    </span>
    <span class="pub">
      <xt:use types="string">Publisher</xt:use>
    </span>
  </p>
</xt:component>

```

Example 3: a component

In this example, note that the two occurrences of the name **refbook** are different. On the first line, **refbook** is the name of the type that is defined by the XTiger element **xt:component**. It is used in the XTiger language every time this type is used, like in example 4. On the second line, **refbook** is the value of an attribute of the target language. More precisely, it is the value of the **class** attribute associated with the **p** element that will instantiate the new type in document instances. These two values could have been different, like in example 1, but template authors often choose the same name for these two notions.

Element **xt:union** is another constructor. It defines a new type as the union of other types. In example 4, the new type **bibref** is defined as the union of the different types of bibliographic citations we have mentioned above.

```

<xt:union name="bibref" include="refbook refarticle
                                refreport"/>

```

Example 4: a union

Defining a union type by enumerating every single type it includes may be very verbose. For that reason, it is possible to use other unions in the list of types (attribute **include**). To make this mechanism more comfortable, element **xt:union** has also an attribute **exclude** which indicates the types that are part of some types listed in the **include** attribute, but that are not accepted for the new type being defined. In other words, the **xt:union** element works in two steps: it first includes types, then it refines this set by removing a few excluded types.

A type defined by a **xt:union** may include types defined by other XTiger elements, as in example 4, but also types from the target language. This is useful to provide flexibility in some parts of a document, where some components defined in the template can be mixed with elements from the target language.

To simplify the writing of templates, a few unions are predefined. Union **any** includes all basic types, plus all types defined by the **xt:component** and **xt:union** elements, plus all types from the target language. The other predefined

unions are more selective: `anySimpleType` includes only the simple types, `anyComponent` includes only the types defined by `xt:component` elements, and `anyElement` includes only the elements from the target language.

3.2 Structure Constraints

In a template, everything specified in the target language must occur as is in all document instances, except when a XTiger element states otherwise. The top level structure of an article like this one (title, authors, abstract, categories, terms, keywords, etc.) is represented in a template simply by its XHTML markup. However, the contents of these elements are obviously not pre-defined. They must then be represented by some XTiger elements.

One of these XTiger elements is `xt:repeat`, which allows a structure to be repeated several times. The structure to be repeated is the content of the `xt:repeat` element. It could be a single XTiger element, or a target language element, or a sequence of elements from both languages. Example 5 shows how the abstract of an article is defined as a `div` that contains a predefined heading `h2` followed by a sequence of `p` elements. The `div` and `h2` elements must appear in an instance exactly as specified in this template (they are expressed in the target language), but the number and the content of the paragraphs are free (this is specified by the XTiger elements).

```
<div class="abstract">
  <h2>ABSTRACT</h2>
  <xt:repeat>
    <xt:use types="p"/>
  </xt:repeat>
</div>
```

Example 5: the repeat element

In this example, the structure to be repeated is a single element of type `p` in the target language (see element `xt:use` below). In many cases the structure is not that simple. Types defined by a component or a union may be used in these cases. Example 6 shows how a more constrained structure can be defined. It specifies the bibliography in the template for an article. After a `h2` element with a predefined content, three to thirty instances of the structure defined by type `bibref` (see example 4) may occur. The number of occurrences of the repeated structure is specified by two attributes, `minOccurs` and `maxOccurs`, which indicate respectively the minimum and maximum number of occurrences. Both attributes have a default value (1 and unbounded resp.) and are optional.

```
<div class="bibliography">
  <h2>REFERENCES</h2>
  <xt:repeat minOccurs="3" maxOccurs="30">
    <xt:use types="bibref"/>
  </xt:repeat>
</div>
```

Example 6: bibliography template

A structure that can be repeated 0 or 1 time is treated as a special case, as authors consider this as an option rather than a repetition. It is represented by an element without any attribute, `xt:option`. In the article example, this element can be used to allow an optional Acknowledgements section to be inserted before the bibliography.

The basic types and the types defined by both elements `xt:component` and `xt:union`, as well as the predefined unions, are used by the `xt:use` element. Wherever this element appears in a template, it indicates what type of element can be inserted at that position.

The `xt:use` element constrains the type of the element that can appear at its position, but its content is free. In example 5, the elements following the `h2` heading must be paragraphs (`p`), but these paragraphs may contain anything that is allowed by their DTD. In example 2, the `span` element representing an address can contain character strings and `br` elements. Note that in example 2 and example 3, several `xt:use` elements have a content in the template. This is an indication of a possible content, a kind of initial value, but it may be changed freely in a document instance. This is different from the content that appears in a target language element, which can not be changed in an instance document: the content of the `h2` elements in example 5 and example 6 must always be ABSTRACT and REFERENCES.

Obviously, it is possible to restrict the content of an element produced by `xt:use`. If the `types` attribute of a `xt:use` specifies a component, the `xt:use` element must be replaced in an instance by the exact structure of the component. Only the sub-elements of the component that are specified by XTiger elements can be different. According to example 3, an element `<xt:use types="refbook">` will be replaced in an instance by a `p` element with an attribute `class="refbook"` and this element will contain one or more `span` elements with attribute `class="bibauthor"` followed by a `span` with `class="title"` and another `span` with `class="pub"`. Only the number of authors and the content of the `span` elements are free.

While `xt:use` is mainly used to constrain the structure, the `xt:bag` element brings some freedom where it is needed. It defines the set of types that can be used in the subtree that will replace it in a document instance. It constrains only the element types, but not the way they are assembled. The `xt:bag` element has a `types` attribute which specifies the allowed element types. This is a list of type names that may contain basic types, unions, components, and element types from the target language. In the article template, we use `xt:bag` to define the content of a section. See example 7 which defines the body of the article.

```
<xt:repeat minOccurs="1">
  <div class="section">
    <h2>
      <xt:use types="string">Heading</xt:use>
    </h2>
    <xt:repeat>
      <xt:bag types="anyElement"/>
    </xt:repeat>
  </div>
</xt:repeat>
```

Example 7: a bag

With this definition, any XHTML element could appear after the initial `h2` in a section. In fact, the definition is a bit more restrictive. Like all other XTiger elements, `xt:bag` is not supposed to violate the structure of the target language. XTiger simply adds more constraints for using the target language. The constraints stated in the DTD or schema of the target language still apply. In the section defined in

example 7, although the XTiger definition of `anyElement` includes XHTML elements such as `head`, `body`, or `meta`, the XHTML DTD makes them unusable within the `div` element representing a section.

Note that elements `xt:use` and `xt:bag` have a `types` (plural) attribute. Its value is a list of type names. All these types can be used to instantiate the `xt:use` element or to fill the `xt:bag` element.

XTiger does not consider only the elements of the target language, but also attributes. Element `xt:attribute` defines rules for using an attribute from the target language. It always appears as a child of a target language element and specifies how to use a given attribute with this element. It may make an attribute mandatory or optional, it may specify a fixed value, a default value, the set of all allowed values, or it may let the value free. It may also prohibit the attribute to be associated with the element. These constraints should not contradict those from the DTD or schema.

```
<xt:component name="author">
  <p class="vcard">
    <xt:option>
      <img class="photo">
        <xt:attribute name="alt"
                    default="Author picture"/>
        <xt:attribute name="width" use="optional"/>
      </img>
    </xt:option>
    <span class="fn">
      ...
    </p>
  </xt:component>
```

Example 8: defining attributes

Example 8 modifies type `author` defined in example 2 by adding an optional photograph of the author (an `img` element in the XHTML target language with class `photo` from hCard) that can be inserted before the author name. The first `xt:attribute` in the example makes the `alt` attribute of the `img` element mandatory and provides a default value for it. The second `xt:attribute` element states that the `img` could have a `width` attribute, but no value is specified. Note that the XHTML DTD already says so, but if this statement was not present, users could not set the width of these images: attributes that are not explicitly mentioned in a template are forbidden in the instance. In the example, nothing is said about the `src` attribute of the `img`, but the XHTML DTD makes this attribute mandatory. It is then mandatory in any instance derived from the template where the component `author` is used. Type `author` does not put any additional constraint on this attribute.

Note that in this example, a `xt:attribute` element could be used to make the `class` attribute mandatory for the `p` element. The second line of example 8 is equivalent to:

```
<p>
  <xt:attribute name="class" fixed="vcard"/>
```

The `xt:attribute` element can apply not only to target language elements, but also to the `xt:use` element. This is useful if we want to let all attributes of the `img` element free (with the only constraints from the DTD), but make a `class` attribute mandatory with a fixed value. To do so, lines 4-8 of example 8 should be replaced by:

```
<xt:use types="img">
  <xt:attribute name="class" fixed="portrait"/>
</xt:use>
```

3.3 Sharing and reuse

Elements `xt:component` and `xt:union` define new types (see section 3.1). In a template, they are grouped at the beginning of the document in a special element, `xt:head`. This element is unique in the template and must appear before using any type it defines.

In addition to type definitions the `xt:head` element may contain some `xt:import` elements, which import external definitions for use in the template. These external definitions are grouped in separate resources called *libraries* which, like element `xt:head`, contain type definitions and `xt:import` elements: a library may import other libraries.

Element `xt:import` has a single attribute that specifies the URI of the library to be included. The order of the `xt:import` elements is used to choose among different type definitions that have the same name: the latest imported library wins. However, types defined in the `xt:head` element have the highest priority.

Libraries allow types that are used in different templates to be shared. They are especially useful for popular microformats. In the article example, types `bibref`, `refbook`, `refarticle`, `refreport` would typically be defined in a library to be used also in templates for reports, theses, and other types of scholar documents.

As we have seen in section 2, templates enable a rich structure that can be exploited by style sheets. Most of the time, templates are developed with a set of accompanying style sheets. Each template includes style information using the mechanisms offered by the target language: `style` element, processing instruction or `link` element to refer to an external style sheet, or even `style` attributes. Like other aspects of a template, this style information is transmitted to the document instances.

4. IMPLEMENTATION ISSUES

XTiger was designed to be implemented in a document editor. It represents generic structures in a way that could be efficiently used by an editor to help authors structure and encode semantic XHTML and microformats. To make the language usable, it was implemented in Amaya [13], a structure editor for authoring XHTML and other XML documents. In this implementation the main issues were structure manipulation and user interface.

4.1 Structure manipulation

In its original version, Amaya provides the editing feature for the target language. It can manipulate the structure of XHTML documents through a direct manipulation style of interface. It does so following the XHTML DTD: all actions of the user are performed under the control of the DTD. As a consequence, documents produced by Amaya are well-formed and valid (in the XML sense) and the user does not have to worry about what is allowed where (more details are available in [13]).

To support XTiger, Amaya has to handle the generic structure defined by the XTiger elements in addition to the XHTML structure. These two structures are tightly mixed, and the editor has to handle them simultaneously. Fortunately, Amaya already supports namespaces. But the

XTiger namespace is a bit special as compared to other namespaces that are implemented in Amaya. It is not a document format, like SVG or MathML, for instance. The XTiger elements are not supposed to be displayed as part of the document. Instead, their semantics must be interpreted by the editor to guide the editing process and to make sure that the editing actions of the user are executed in accordance with the XTiger generic structure.

Amaya works in two steps while editing. It first processes the document structure as if XTiger elements were not present. Thus, it can take the DTD of the target language into account. Then it looks at the XTiger elements that are ancestors of the element of interest and further restricts the operations allowed by the DTD according to the semantics of these XTiger elements. When it is in a pure target language part of the document, editing is just locked.

When a user wants to create a document from a template, the new document instance is created as a copy of the template. However, the `xt:head` element with its type definitions is kept by the editor, but it is not copied in the document instance, as type definitions are intended to be shared. The template is linked to the new instance by a processing instruction inserted at the beginning of the instance, in the same way CSS style sheets are linked to XML documents. With this link, the editor will find all the type definitions needed during the subsequent editing sessions. All other XTiger elements (`xt:use`, `xt:bag`, `xt:repeat`, `xt:option`, `xt:attribute`) as well as all target language elements are kept in the copy that constitutes the initial instance. XTiger types that appear in these elements are replaced by references to their definition in the template (actually, by references to a parsed representation of types in core memory which is more compact).

When the initial instance has been created, the user can see a skeleton of the document to be written (see Figure 1). If style sheets are associated with the template, they apply to the skeleton. Then the user interacts with the editor to develop the structure and provide contents. This is done following all constraints, those expressed by the XTiger elements and those of the target language expressed by its DTD. When creating new elements, both target language elements and XTiger elements are created after the template. For instance, when adding a new element in the `xt:repeat` that allows several sections to be created in an article, the whole template of the new section is created (see example 7). This allows the editor to know what is allowed by the template and what is not, just by looking at the current position in the document: the local XTiger elements express the local constraints. As a consequence, the edited document can be seen as a template that grows under user's control.

Guiding the user and checking the document structure is basically what Amaya does natively for the target language, but the difference with XTiger is that the generic structure described by XTiger elements is not separated from the document being edited, like a DTD or a schema. It is part of it. This greatly simplifies structure editing, as there is no need to first go to the DTD and find the rules that are relevant to the current position. The counterpart is that the size of the structure rises, but only in the parts where a strong control on the structure is exercised. For example only two elements (`xt:repeat` and `xt:bag`) are used to control the whole content of a section (see below), while more elements are involved in an author or a bibliographic citation. Note

that all XTiger elements can also be removed when the document is finished (see section 4.3).

On the other hand, the XTiger elements and attributes that are interspersed in the document have to be processed in a special way. When editing a document instance, they are not supposed to be modified by the user, whereas a MathML expression can be modified within a XHTML page.

As an example, consider how a new section may be added in an article. According to example 7, the editor is allowed to create a section as a child of the `xt:repeat` element. Following the definition of example 7, it creates this structure:

```
<div class="section">
  <h2><xt:use types="string">Heading</xt:use></h2>
  <xt:repeat>
    <xt:bag types="anyElement">

    </xt:bag>
  </xt:repeat>
</div>
```

Notice that it copies the `xt:bag` and `xt:use` elements from the definition, thus developing the template and indicating what is allowed at each position. The content of the `xt:use` element is an ordinary string, "Heading", that can be edited freely. Only a string can go in the `xt:use` element, as stated by its `types` attribute, and no sibling element can be inserted before or after the `xt:use` within the `h2`, which is a target language element.

The generated `xt:bag` element tells the editor that it is allowed to create any element of the target language in it, at the position of the empty line. If this element was not generated, the editor could not create anything there, for the same reason it can not create anything before the `h2` in the `div`. The generated structure for the new section being part of the template can not be modified where it is not explicitly stated, like in the `xt:use` or `xt:bag` elements.

4.2 User interface

Even if XTiger elements are not supposed to be displayed in the same way as elements from the target language, it is useful for the user to see them, in order to know what is allowed at each position in the document being edited. A visual representation was chosen for this purpose. Figure 1 shows how a document instance built from (a simplified version of) the article template is displayed on the screen.

When editing a document instance, the user does not need to see the `xt:head` and `xt:library` containers, nor their contents, the `xt:component` and `xt:union` which are type definitions. These elements are not part of the document instance anyway. The only XTiger elements visible on the screen are those constraining the structure: `xt:use`, `xt:bag`, `xt:repeat`, and `xt:option`.

These four elements are represented by dotted boxes that delineate the content of the element, with a small icon in the top left corner. A different color and a different icon are used for each element: blue and triangle for `xt:use`, green and 'U' for `xt:bag`, purple and '+' for `xt:repeat`, and yellow and tick sign for `xt:option`. Note that the colors used in Figure 1, as well as line style, may be modified by the user through CSS style sheets, to avoid any confusion with frames drawn around elements of the target language. This can also be used to make sure that the colors fit well with the background.

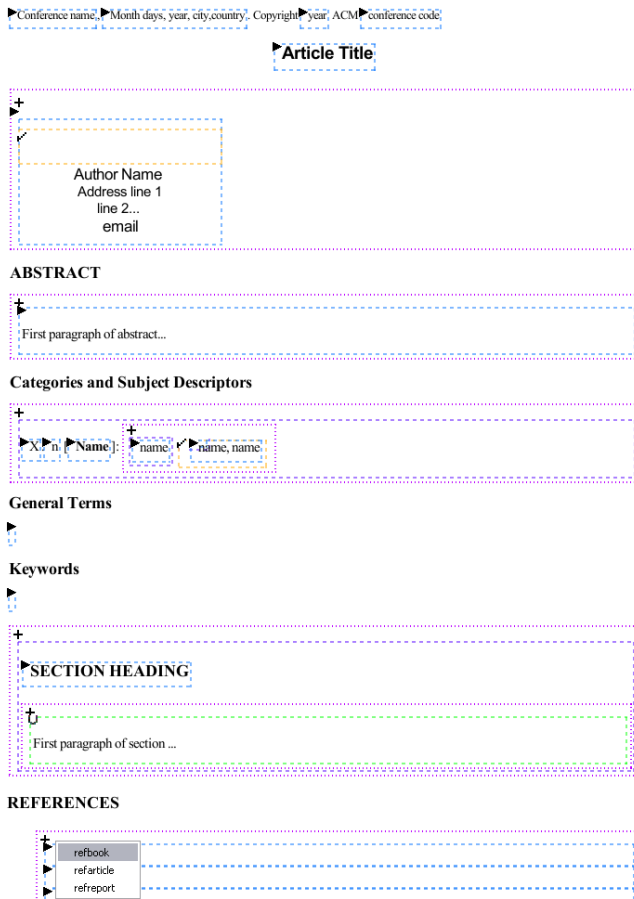


Figure 1: visualization of XTiger elements

The areas identified by dotted boxes are the regions of the document where the user can make changes. All other parts of the document are locked: editing is not allowed there. For example, in Figure 1, headings "ABSTRACT" and "REFERENCES" are not framed and therefore can not be modified by the user (see example 5 and example 6), and nothing can be inserted before or after these headings.

Element `xt:attribute` is not visible on the screen. Its role is only to restrict the use of the attributes defined in the target language. For this reason, its only perceivable effect for the author is a restriction of the usual attribute menu. Attributes that are not allowed per the `xt:attribute` elements are not selectable in the attribute menu, and values that are not allowed can not be entered.

The user needs not only to visualize the XTiger elements, but also to interact with them. She needs to create or remove elements in a `xt:repeat`, to create or delete the content of a `xt:option`, to insert the content of a `xt:use`, to create, modify or delete elements in a `xt:bag`. Some of these operations can be done with the usual user interface of Amaya. For instance, deleting an element in a `xt:repeat` or `xt:option` is done with the usual Delete command (however, the Delete command was extended to prevent the user from deleting elements that are made mandatory by their parent XTiger element). In the same way, creating a new element is done with the usual Enter key, which inserts a new element of the same type after the selected element.

However, a specific user interface was needed to allow users to perform other operations related to XTiger elements. Clicking the small icon in the top left corner of an XTiger element performs different actions, depending on the icon, i.e. the type of the element:

- `xt:repeat`: the '+' sign creates a new instance of the repeated structure before the existing instances (to create a new instance after an existing instance, the standard command of Amaya is used: selecting the existing instance and hitting the Enter key).
- `xt:option`: the tick sign creates the optional element if it does not exist, or it deletes it if it exists.
- `xt:use`: the triangle displays a menu with the different types of elements that can be created. For instance, the menu displayed at the bottom of Figure 1 is built according to example 6 and example 4.
- `xt:bag`: the 'U' sign displays a menu with the different types of elements that can be created. Only types defined by `xt:component` and `xt:union` are listed in this menu. Element types from the target language are created using the usual commands of Amaya. According to example 7, the 'U' sign of Figure 1 does not display any menu, as only elements from the target language are allowed there.

The menus displayed for elements `xt:use` and `xt:bag` are built straightforwardly from the `xt:union` elements. If `xt:union` elements refer to other `xt:union` elements, this creates cascades of menus.

4.3 Editing and using documents

At anytime during an editing session, the document can be saved. It is saved entirely, with all the XTiger elements derived from the template. Thus the document can be loaded later and edited again with all the services offered by XTiger. This is made easier by the processing instruction which links an instance with its template (see section 4.1).

Documents edited with a XTiger template and saved by Amaya may be used with other applications, in particular with browsers. The XTiger elements that Amaya makes visible on the screen and that are used to activate editing commands, are ignored by other applications. Documents are simply displayed as if the XTiger elements were not there. In addition, Amaya offers an option for displaying the document without the graphical representation of XTiger elements: the document can then be seen in exactly the same way as in any browser.

When a document is finished, the XTiger elements it contains are no longer necessary (they are used only during editing). To cope with this situation, Amaya offers a new command that filters out all XTiger elements from a document. The document becomes then a usual target language document, but its structure still conforms to the model described by the template it used before. If the document has to be edited further, it will be considered as a plain target language document. To get back to XTiger editing, the document should be parsed against its XTiger template to reinsert the appropriate XTiger elements. This is considered for further extension.

5. RELATED WORK

The XTiger language and its implementation in Amaya are related to several aspects of structured documents. They were inspired by previous work in different areas of document engineering. The most influential are template languages, structure definition languages, transformation languages and document editors.

5.1 Template languages

XTiger is a template language. Template languages are used for processing structured documents in different ways. In information mining they assist structure recognition. Web servers use them to generate dynamic web pages. They help editors to suggest or impose a model for the documents authors create. All these applications use different kinds of templates and different template languages have been designed in each category.

In information mining, templates are used to drive the recognition process. Compared with XTiger, they have a very different purpose. They help to identify the areas of interest in a document, whereas XTiger helps to create documents. They actually have not much in common.

Templates used on web servers are closer to XTiger. They dynamically generate web pages from various sources of data, such as databases and XML files. Like XTiger, they are designed to produce documents, but with a significant difference: they operate in automatic mode, without human supervision. Instead of describing all the possibilities a user may choose from, like XTiger, they implement algorithms that make a decision in all possible cases. For that reason, most of them are closer to programming and scripting languages. XTiger, with its declarative form, is very different from these imperative languages. Typical server-side templating languages are Python and php, which are supported by various template engines (Cheeta, Smarty, Zope, etc.).

Among these engines, Zope is especially interesting, as it uses an XML syntax, like XTiger. Its templates (ZPT) use the Template Attribute Language (TAL), which takes the form of an XML namespace containing only attributes to be added to the tags of the target language. The TAL attributes express statements like `tal:content`, `tal:repeat`, `tal:condition`, `tal:replace`, `tal:attribute`. These statements are complemented with a query language (a sort of very simple XPath language) that gets from the source data the contents to be generated or the parameters of the statements. Some of the TAL statements are close to XTiger elements, but the main difference is still the automatic process that generates pages.

Most popular HTML editors also offer a template feature (Dreamweaver, FrontPage, etc.), whose functionality is very close to XTiger in its purpose: helping authors to structure HTML documents following a pre-defined model. Some have a very basic notion of a template. They just propose a series of typical documents that can be freely edited and adapted without any dedicated mechanism or language. Some others support a template language that is interpreted by the editor. In most cases this language is embedded in XML comments whose contents must follow a specific syntax. In Dreamweaver, for instance, these special comments can prevent the author from modifying some parts of the template document; they can allow her to repeat a given part (possibly 0 times); they can finally let some parts completely free.

Compared with this kind of language, XTiger shows two main differences. First its syntax is XML, which provides better control on its integration with the target language. With an XML syntax, well-formedness rules allow to check for instance that a piece of structure in the target language is correctly embedded in a template language command. Second, the expressive power of the template language in XTiger is higher. XTiger provides a fine-grained mechanism for controlling the structure of document instances. It is able to set different levels of constraints on the contents of a document. Dreamweaver, by contrast, only allows the content of a document to be kept unchanged, to be repeated unchanged or to be totally free. This works well as long as only traditional web pages are created, but this is not flexible enough for producing semantic XHTML and microformats.

XTiger is also more powerful in the sense that the target language is not restricted to XHTML. XTiger can be used with any XML document format. When editing mathematical expressions in MathML [4], complex structures that are used repeatedly may be defined as XTiger components. When creating graphics in SVG [8] with a regular structure, shapes that occur multiple times may also be pre-defined. This extends the notion of microformats, initially developed for XHTML, to other languages. This also makes it possible to define libraries of predefined, adaptable structures that greatly help authoring scientific or graphic documents. Note however that a language like SMIL [3] is not considered here. Even if SMIL is a multimedia document representation language, it does not represent the logical structure of a document, but rather its temporal structure. XTiger could only help to predefine some temporal structures, which is not what multimedia authors are usually expecting primarily from a template language.

XTiger can also help editing compound documents, i.e. documents using several document representation languages such as XHTML, SVG and MathML. Its implementation in Amaya exploits this feature, as Amaya provides support for compound documents [14].

5.2 Structure definition languages

XTiger is a structure definition language. In this category we find DTDs, XML Schema [7] or Relax-NG [6]. XTiger borrows some of its features from these languages. The definition of types with elements `xt:component` and `xt:union` is derived from XML Schema. Element `xt:attribute` is almost the same as in XML Schema. Attributes `maxOccurs` and `minOccurs` for the `xt:repeat` element are very close to the attributes of same name in XML Schema. As mentioned in section 3.1, basic types in XTiger are compatible with the datatypes in XML Schema.

On the other hand, schema languages are different from XTiger. Their purpose is more general: they describe all possible instances of a document type, and they do not target a particular type of application, whereas XTiger was designed only for editing. Schemas can be used to guide an XML editor [15], but they are also used in different kinds of applications where validation is required.

Also, XTiger elements include a lot of predefined content, which is an important feature for a template language. In this regard XML Schema is very different. XTiger is simpler than schemas, because it is used only for expressing additional structural constraints, building on top of the target language, which is itself defined by a schema or a DTD.

5.3 Transformation languages

XTiger borrows from transformation languages: some aspects of its design were influenced by XSLT [5]. Both XTiger and XSLT are based on the notion of a template. In both languages the template is expressed in the target language. Both languages are mixed with the target language and use the namespace mechanism to separate languages and avoid ambiguity.

An important difference however is that XSLT operates in automatic mode, like server-side templating languages. In XSLT, XPath expressions are used to select the source information to which a template has to be applied. XTiger is not a transformation language. So, there is no source information to be located before applying a template. The template is already at the position where it has to be used in the target document, and the author provides the contents and chooses between the various options offered by the template. As a consequence, only the generation part of XSLT could be compared with XTiger; there is no equivalent to the selection part.

As opposed to schema languages (see section 5.2), XTiger was not designed to validate documents. However, its implementation in Amaya makes sure that documents are always valid in the target language. In this regard, it is quite different from XSLT, which can not guarantee that the result of a transformation is a valid document.

5.4 Document editors

XTiger was designed to facilitate document editing with semantic markup. Its implementation in Amaya can then be compared with HTML and XML editors. Some aspects of this comparison have already been addressed in the discussions of template languages (section 5.1) and structure definition languages (section 5.2), but a few other aspects of editing are worth being considered.

Most XML editors are based on DTDs or schemas [9]. They check validity during editing. But XML documents may also exist independently of any DTD or schema, with the only constraint of being well-formed. Amaya can edit this kind of documents [14]: it allows authors to create XML structures without any constraint on the type of elements and their attributes. To take full advantage of this feature, it appears that a light structure definition with a XTiger template is a great benefit for authors who need some help to structure such documents. Used that way, XTiger fills the gap between a completely free structure that has just to be well-formed and the very strong constraints imposed by a DTD or a schema.

One of the motivations for developing XTiger was to provide a tool for editing microformats. Most of the tools currently available for that task are quite basic. They are usually form-based, like some XML entry user interfaces [11]. The author fills up a form, and the tool generates the XHTML markup that encodes this information in the microformat. The generated markup has then to be included in the source code of the web page where the microformat has to reside. hCard Creator [16] and XFN Creator are typical examples of this kind of tool. With XTiger, the most popular microformats may be described in libraries and reused in many different types of documents, themselves described as XTiger templates. Authors can then edit microformats directly within the document, with the same tool they use for editing the rest of the document. Making changes to

information that is already present in a document is as easy as creating it, while the form-based tools do not help in that situation.

6. DISCUSSION

In previous sections, all examples are extracted from the XTiger template that was used to produce this article. The full template can be found at <http://wam.inrialpes.fr/publications/2006/DocEng/ACM-Proc-Article.xtd>. The final document, built from this template, is available at the same location under two different forms: with the whole XTiger markup (DocEng2006full.html) that allows the document to be edited according to the template, and without that markup (DocEng2006.html), as a plain XHTML document, for final publication. The ratio of the sizes of both files is 1.26.

This figure has to be compared with the same ratio for initial documents, i.e. the minimum instance built when creating a new document from the template (see Figure 1). In this case, the ratio is 1.60. This is an extreme case. In an initial document there is almost no content, and the XTiger markup takes a large part of the document.

It should be noted however that the microformats used for the authors and for the bibliography are very simple. This is for the sake of simplicity of examples 1 to 4, and also because there is not yet a widely approved microformat for bibliographic citations.

For publication in the conference proceedings, and to make sure the printed version fully adheres to the publisher's rules, the XHTML document was transformed into a LaTeX file that makes use of the style provided by the publisher. The transformation was written in XSLT, and this was very easy. It is known however that developing a transformation sheet from XHTML to LaTeX is a huge task, and the existing transformation sheets are very complex. In addition, they have to be adapted for each specific LaTeX style. In our case, the structure of the XHTML file is constrained by the template which considerably restricts the possible structures that have to be handled. Also, the correspondence between this constrained XHTML structure and the specific commands from the LaTeX style is quite simple, as the template was designed to cope with this style. As a result, the XSLT sheet is only 249 lines long.

It should be noted that the markup representing authors in this article uses only a very small subset of the hCard microformat, and this subset is used with a specific structure, while hCard specifies 45 different classes which may appear in almost any order and any structure. But XTiger is not a schema language. Its goal is not to describe all the possible ways of using a microformat, but to define a way of using it in documents of a certain type. For instance, the XTiger template for CV's (based on the hResume microformat) uses a bit more classes from hCard, to give more detailed contact information about a person. The strength of XTiger in this context is that it allows to constrain both the overall structure of the document and the use of various microformats within that document.

7. CONCLUSION

XTiger was designed to help authors to create semantically rich web pages based on the concepts of semantic XHTML and microformats. It is an original language that

simultaneously presents features usually offered separately by template languages, schema languages, and transformation languages. Despite this wide scope, the language is very simple. In the current version it contains only nine elements and each element has a very limited number of attributes. These elements are used to define types (`xt:component`, `xt:union`), to define containers for type definitions (`xt:head`, `xt:library`), and to specify the allowed elements and attributes at various places in the document structure (`xt:use`, `xt:bag`, `xt:repeat`, `xt:option`, `xt:attribute`).

Such a limited vocabulary was made possible because the XTiger language is used only to define a specialized use of an existing, more general language, the target language. However, this minimalist language is powerful enough to describe both the general organization of large documents and the fine-grained structure of microformats. Being defined on top of an existing language (typically XHTML), these structures may be used and deployed very quickly, taking advantage of the infrastructure available for the target language.

XTiger is implemented in the Amaya editor. This implementation is also reasonably simple, as it is based on an existing editor that already manipulates the target language(s). Implementing the editing features associated with XTiger on this basis was not a huge task. The result is a powerful yet easy-to-use tool that efficiently helps authors to manipulate documents according to the exact model they need for each type of document. The tool guarantees that the documents produced follow the chosen model (template) and that they are always valid regarding the target language(s). Documents can then be used with any tool that supports the target language(s). This includes obviously web browsers when the target languages are XHTML and SVG, for instance.

Although the current version of XTiger and its implementation in Amaya are already offering useful services, a few extensions may be envisioned. A useful addition would be a tool for creating templates. Currently, a new template is created in two steps. First, a skeleton in the target language(s) is edited using only the initial functionality of Amaya. Then, XTiger elements are added "by hand" to make a template. This second step could be assisted in Amaya, by supporting the editing of XTiger elements and attributes in a specific mode, where these elements would not be interpreted, but just edited like other XML elements and attributes.

Finally, the current tool does not provide any help when templates have to be modified. In particular, when a template changes, it would be helpful to have a tool that could update existing documents, taking into account the new structure of the template. Further investigations are considered to address this issue.

8. 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.

9. REFERENCES

- [1] T. Bray, D. Hollander, A. Layman, and R. Tobin. *Namespaces in XML 1.1*. W3C Recommendation, <http://www.w3.org/TR/xml-names11/>, 4 February 2004.
- [2] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Third Edition)*. W3C Recommendation, <http://www.w3.org/TR/REC-xml/>, 4 February 2004.
- [3] D. Bulterman, G. Grassel, J. Jansen, A. Koivisto, N. Layaida, T. Michel, S. Mullender, and D. Zucker. *Synchronized Multimedia Integration Language (SMIL 2.1)*. W3C Recommendation, <http://www.w3.org/TR/SMIL/>, 13 December 2005.
- [4] D. Carlisle, P. Ion, R. Miner, and N. Poppelier. *Mathematical Markup Language (MathML) Version 2.0 (Second Edition)*. W3C Recommendation, <http://www.w3.org/TR/MathML/>, 21 October 2003.
- [5] J. Clark. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation, <http://www.w3.org/TR/xslt>, 16 November 1999.
- [6] J. Clark and M. Murata. *RELAX NG Specification*. Oasis, <http://www.oasis-open.org/committees/relaxng/spec-20011203.html>, 3 December 2001.
- [7] D. Fallside and P. Walmsley. *XML Schema Part 0: Primer, Second Edition*. W3C Recommendation, <http://www.w3.org/TR/xmlschema-0/>, 28 October 2004.
- [8] J. Ferraiolo, J. Fujisawa, and D. Jackson. *Scalable Vector Graphics (SVG) 1.1 Specification*. W3C Recommendation, <http://www.w3.org/TR/SVG/>, 14 January 2003.
- [9] R. Furuta, V. Quint, and J. André. Interactively editing structured documents. *Electronic Publishing Origination, Dissemination, and Design*, 1(1):19–44, April 1988.
- [10] R. Khare. Microformats: the next (small) thing on the semantic web? *IEEE Internet Computing*, 10(1):68–75, 2006.
- [11] Y. Kuo, N. Shih, L. Tseng, and H.-C. Hu. Generating form-based user interfaces for XML vocabularies. In *Proc. 2005 ACM Symposium on Document Engineering*, pages 58–60. ACM Press, November 2005.
- [12] H. W. Lie and B. Bos. *Cascading Style Sheets, Designing for the Web, 3rd edition*. Addison Wesley, 2005.
- [13] 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.
- [14] V. Quint and I. Vatton. Towards active web clients. In *Proc. 2005 ACM Symposium on Document Engineering*, pages 115–123. ACM Press, October 2004.
- [15] M. Sifer, Y. Peres, and Y. Maarek. Browsing and editing XML schema documents with an interactive editor. In *Proceedings of DNIS 2003, LNCS 2822*, pages 97–111, September 2003.
- [16] T. Çelik. *hCard Creator*. <http://tantek.com/microformats/hcard-creator.html>, 2005.
- [17] T. Çelik, E. Meyer, and M. Mullenweg. XHTML meta data profiles. In *Proc. 14th International World Wide Web Conference (WWW05)*, pages 994–995. ACM Press, 2005.