Authoring XML all the Time, Everywhere and by Everyone

Stéphane Sire, Christine Vanoirbeek, Vincent Quint, Cécile Roisin

Authoring XML all the Time, Everywhere and by Everyone

Stéphane Sire
EPFL
<stephane.sire@epfl.ch>

Christine Vanoirbeek
EPFL
<christine.vanoirbeek@epfl.ch>

Vincent Quint
INRIA
<vincent.quint@inria.fr>

Cécile Roisin
INRIA
<cecile.roisin@inria.fr>

Abstract

This article presents a framework for editing, publishing and sharing XML content directly from within the browser. It comes in two parts: XTiger XML and AXEL. XTiger XML is a document template specification language for creating document models. AXEL is a client-side Javascript library that turns the document template into a document editing application running in the browser. This framework is targeted at non XML speaking end users, since it preserves end users from XML syntax during editing. Its current implementation proposes a pseudo-WYSIWYG user interface where the document template provides a document-oriented editing metaphor, or a more form-oriented metaphor, depending on the template.

Keywords: XML, authoring, document template, client-side Javascript library

1. Introduction

Nowadays, most web-based applications take advantage of the XML format to expose information on the web – by use of XSLT transformations and/or CSS style – or, still better, to anchor automatic processes on XML data that follow a conceptual model. Despite the fact that XML originates from research in the structured document do-

main – XML is an application profile of SGML – , the common use of XML on the web is mostly based on data extracted from relational databases. In this sense, web users essentially contribute to populating such databases by introducing data through forms displayed on their browser. On another hand, the semantic web perspective reinforces the trend to produce more document-oriented information as it encourages to provide information that embarks significant meaning by the use of tags and attributes employed in documents. As a consequence XML authoring should address both paradigms.

The use of forms constrains users to provide consistent data-oriented information. It may be performed by using XHTML forms; in this case, a significant effort is required on the server side to check validity of entered data. It may be performed by the use of XForms whose declarative approach clearly facilitates the control of data, but unfortunately XForms is currently not supported by browsers, and thus requires the use of often complex frameworks.

Producing document-oriented information on the web is feasible but is currently addressed in so many different ways: web-based rich text editors, Wikis and blogs, XHTML editors or Content Management Systems. Web-based rich text editors allow web users to provide XHTML content in a way very close to usual desktop word processors. Wikis or blog authoring systems allow users to produce tagged information but, often necessitate the knowledge of a basic syntax which is not appropriate to end-users. Most XHTML editors offer the possibility to introduce micro-formats in the documents, leveraging the level of reusability of common components. Content management systems offer functionality to produce either XHTML or XML content on the web. More recently, web-based WYSIWYG XML editors became available.

As a result, millions of (X)HTML blog entries, wiki pages or database generated content are currently available, but they are difficult to interpret and repurpose. XML content is also available, but XML-based authoring solutions imply using complex client/server architectures. They include the use of XML parsers to validate information against a generic model (DTD, XML Schema, Relax-NG, etc.), usually on the server side.

Imagine what would be possible if non XML-savvy users could use their familiar browser as an XML authoring tool, allowing them to produce mixed data- and document-oriented information, constrained by a specific template. As an extension, with the agreement on common XML vocabularies or domain specific languages, all data entered on the web would be available for automatic processing [12]. Moreover, the use of common XML vocabularies would allow to develop standard editing components. This modularity would allow to quickly create light XML authoring applications adapted for specific content models.

This article presents AXEL (Adaptable XML Editing Library), a client-side Javascript library for authoring template-driven XML content on the Web. It relies on the use of XTiger XML, a template definition language based on XHTML, which is designed to express structural constraints on the edited content. Additionally,

the associated use of CSS provides flexibility in terms of user interface. The template language and the library bring the opportunity to provide users with traditional forms-based interfaces or document-oriented interfaces, allowing them to author XML content with a visual user experience close to WYSIWYG word processors.

The paper is organized as follows: we introduce first the concept of template proposed by XTiger XML and explain its articulation with AXEL, the client-side document template engine that transforms a template into an interactive XHTML page. Then we provide details about the XTiger XML syntax and semantics and we illustrate through concrete examples how templates constrain the document structure, to guide the presentation and to support the mapping to a target XML structure. Then we describe the editing functionality provided by AXEL and justify the customization of user interface in terms of usability. Finally, we explain some server integration aspects and we outline the main features of the library. The article finishes with a comparison of our approach with related work and some perspectives for future work.

2. Templates at a Glance

2.1. Document Templates

In many document production systems such as word processors, templates are typical documents whose organization and style indicate how documents of a certain type should be structured and presented. With this approach, a template is just a sample document that authors use as a starting point and that they develop by providing content while following the structure and style of the initial document. The tool lets authors free to change everything at any time, but the initial sample document is supposed to give them hints about what is expected in the end. In some systems, in XHTML editors for instance, templates contain also some fixed parts that an author can not modify and that will be preserved in the final document, to strongly constrain some parts of the document.

Different types of templates are also used in content management systems for generating HTML pages from a data base. In that case, the template is a HTML page with "holes" that contain queries for extracting content from the data base. Some statements are also interspersed in the HTML code to generate additional parts depending on the data found in the data base.

So, there are different sorts of document templates. The templates we are considering in this paper were initially designed [7] to make it easier for authors to create and edit well structured and semantically rich XHTML documents that do not require complex schemas and transformations, while still allowing documents to provide useful, automatically processable information. The initial language we have developed for this purpose was called XTiger [11].

The idea behind XTiger was to use XHTML as the basic document format and to constrain the way to use it, in order to produce a specific type of document. A number of authors are comfortable with (X)HTML editors, and some of these editors produce valid, well structured XHTML code. Because XHTML is an XML language, the namespace mechanism can be used to allow XHTML documents to include elements and attributes from another XML language, XTiger, that expresses constraints on the XHTML structure. As XHTML can benefit from CSS style sheets, it is easy to specify the visual aspect of the document structure.

With this approach, a template is an XHTML document (with its style sheets) that represents the skeleton of a document, and that contains statements expressed in the XTiger language to indicate how this embryonic document can evolve and grow. We have extended the Amaya web editor [1] to support the original XTiger language. The editor interprets the XTiger statements contained in the template to guide the user in building the intended document structure. The author interacts with a familiar editor on a well formatted document and eventually produces a structured XHTML document, that can then be used directly on the web with any browser. Templates are easy to create: they are basically XHTML documents containing XTiger elements where structural constraints have to be set.

This approach to templates has proven to be very useful for editing rich XHTML documents on the web [7], but it is possible to go a step further and to produce any XML structure, not only XHTML, while preserving the ease of use of XTiger templates. This is done by adding to the original XTiger language a mapping mechanism for specifying what piece of XHTML structure is equivalent to what target XML structure. Another significant step is to let authors free to choose their preferred tool, by implementing the editor as a Javascript library that runs in a browser. When loaded into the browser with the library, the document template is turned into an interactive editor that follows the constraints expressed by the XTiger elements to generate only data following a predefined XML content model.

This extended version of the template language is called XTiger XML [14]. In the remainder of this paper, we often write XTiger when discussing features offered by both versions of the language.

Thus, an XTiger XML template plays three complementary roles:

- It includes constraints that define the data components the user can enter at different places in the document.
- It contains presentation hints (XHTML elements, style sheets) that define how each editing component looks like while editing.
- It provides information for mapping the editing components to a target XML structure.

The figure below shows a typical XML editorial process built with document templates: a template author creates the templates which are turned into interactive editing applications into the browser by the AXEL library. End users can then create

or change documents; when they save them, their XML content is generated and sent to server side applications for further processing or stored into databases or documents.

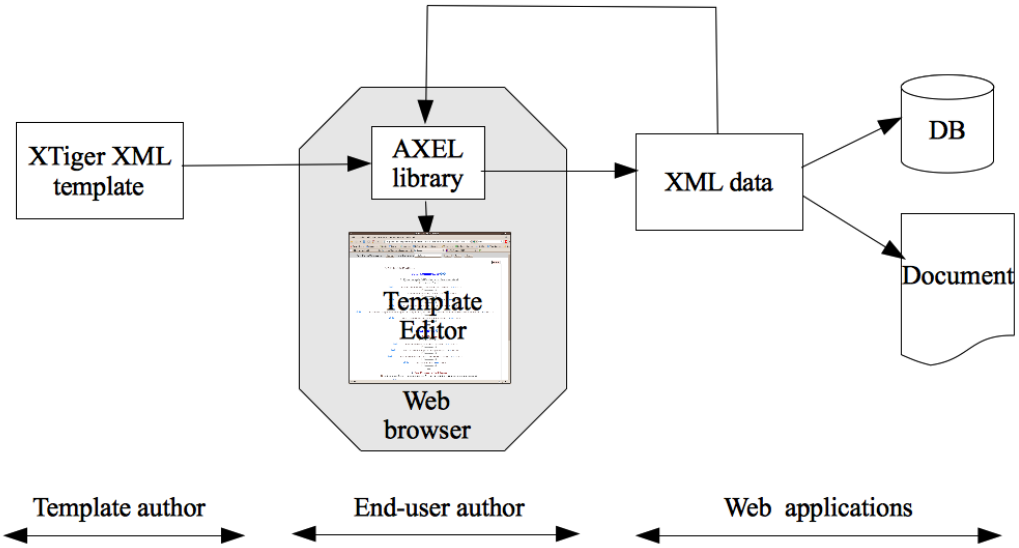

Figure 1. Overall architecture

2.2. Anatomy of an XTiger XML Template

The following code example shows a very simple document template for editing a list of persons to great. As this section explains, the document template defines in the same file: some structural editing constraints, a presentational view for editing data, and an XML content model.

Example 1. The "Greetings" template

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xt="http://ns.inria.org/xtiger">
<head>
 <title>My first template</title>
 <xt:head label="greetings">
   <xt:component name="friend">
     <li><xt:use types="text">name</xt:use><xt:menu-marker/></li>
   </xt:component>
 </xt:head>
</head>
<body>
 <p>List of persons to great:</p>
 <ul>
```

```
    <xt:repeat minOccurs="0" maxOccurs="*" label="persons">
      <xt:use types="friend" label="name"/>
    </xt:repeat>
  </ul>
</body>
</html>
```

A capable document template engine, such as AXEL, transforms the previous template into an interactive XHTML page from which a user can create a list of persons. At any time, the page content can be exported into XML, as in the following example.

Example 2. Example of edited XML content

```
<greetings>
  <persons>
    <name>Charlie</name>
    <name>Oscar<name>
  </persons>
</greetings>
```

The "Greetings" example shows that the document template is an XHTML document. This is because, as it will be run inside a browser, the presentation language is the language for displaying pages in the browser. The document also contains some elements and attributes in the `http://ns.inria.org/xtiger` namespace which is prefixed with `xt:` for XTiger. These elements constrain user's input and define an XML content model.

The document template is divided into two parts. The `xt:head` part, which is declared in the `head` section of the XHTML document, declares some reusable editing components and data types. Each component is declared as a `xt:component` element with a `name` attribute that will be used to refer to it. The content of a component can mix elements from the presentation language, XHTML, and from the XTiger namespace.

The second part of the template is the `body` section of the XHTML document. It contains a mix of XHTML elements and XTiger elements. The XHTML elements are rendered as usual within the browser. They give the document its appearance and can be styled using CSS to generate different looks and feels. The `xt:use` element is a component inclusion element which makes it possible to insert an editing component in place and specify its data type. The document template engine expands each `xt:use` element by replacing it with the content of the component whose name matches the value of its `types` attribute. Ultimately, some editing component names such as the `text` component are reserved types which are associated with *primitive editor components*. The primitive editor components are transformed by the engine into special fields which can be edited by the end user to enter data in the document.

Some other XTiger elements and attributes, such as the `xt:repeat` element and its `minOccurs` and `maxOccurs` attributes are editing constraints that guide the editing process. The `xt:repeat` element indicates that its content can be repeated several times, with a minimum and a maximum number of times.

Finally, the XTiger attribute `label` drives the XML content serialization process. In fact, the edited document is turned into XML content by traversing it from its root. Each time the template engine encounters a `label` attribute, it creates a new XML content fragment with a tag name set to the value of `label`. The XML content of this fragment is the result of serializing the corresponding document subtree.

2.3. Editing With a Document Template

A client-side document template engine, such as AXEL, loads a document template in a browser window, and transforms it into an interactive editing application. As can be seen on Figure 2, some parts of the document template are transformed into user interface controls such as a *minus* and a *plus* button. Some other parts of the document become editing fields, such as the fields to enter a person name. Finally, some parts are just part of the background and are displayed as usual, non-editable XHTML elements, such as the list header in the snapshot below.

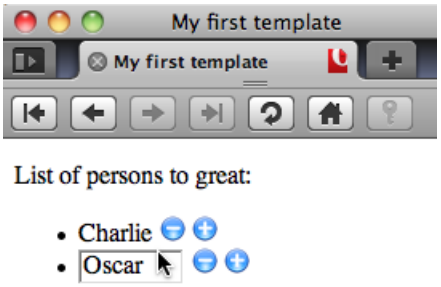

Figure 2. The "Greetings" generated editor

The user interface generated by the document template engine is quite different from a *Rich Text Editor* user interface. The main reason is that there is no need for a command panel to group all the available options in one place (usually at the top of the window), because the template prevents users to insert document formating commands. Instead, an editor inserts directly into the document some user interface controls (such as the *minus* or *plus* buttons) directly within the document flow, at the position where the choice is available to the user.

The areas where users can input text are defined by primitive editor components. Each of these editors is responsible to manage the display and the editing of its content. For instance, the `text` primitive editor component displays its content either as an XHTML `span` element, or as an `input` or `textarea` that dynamically replaces

the content when the user clicks on it for editing. This is illustrated with the *Charlie* and *Oscar* inputs in the figure above.

In order to preserve a document look and feel, the document template editor does not systematically emphasizes the editing component borders, for instance with dashed boxes, as it is the case in other document template editors such as Microsoft PowerPoint. Instead, each document template author can use CSS properties and `class` attributes to create the desired effects.

To some extent, the editing user interface can be seen as something between a full WYSIWYG editing user interface and a more constrained form-based user interface. In any case, it never shows XML tag or attributes to the user, which makes it easily usable by everyone.

The editing user interface is self-contained in the document (no need for extra menus or panels) and the document template engine implementation is client-side. As a consequence, editable document templates can be embedded anywhere in any application (see section on server integration for details). For instance, for the purpose of writing this article, we have deployed the document template engine onto a WebDAV web server. The only addition to the document template of the article (which uses a custom XML content model from which we can easily generate a Docbook or an XHTML document), has been to program a Javascript menu bar to load and save XML data from and to the document template. The resulting editing application with its menu bar is displayed below:

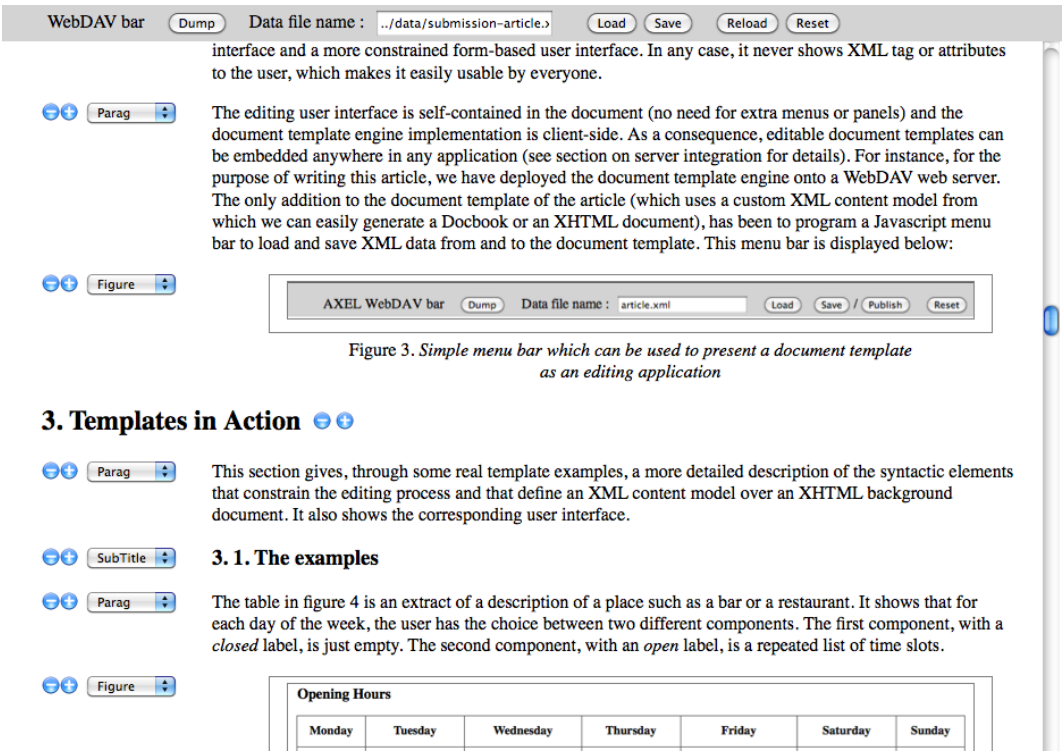

Figure 3. A simple editing application with it's menu bar for authoring this article

3. Templates in Action

This section gives, through some real template examples, a more detailed description of the syntactic elements that constrain the editing process and that define an XML content model over an XHTML background document. It also shows the corresponding user interface.

3.1. The examples

The table in Figure 4 is an extract of a description of a place such as a bar or a restaurant. It shows that for each day of the week, the user has the choice between two different components. The first component, with a *closed* label, is just empty. The second component, with an *open* label, is a repeated list of time slots.

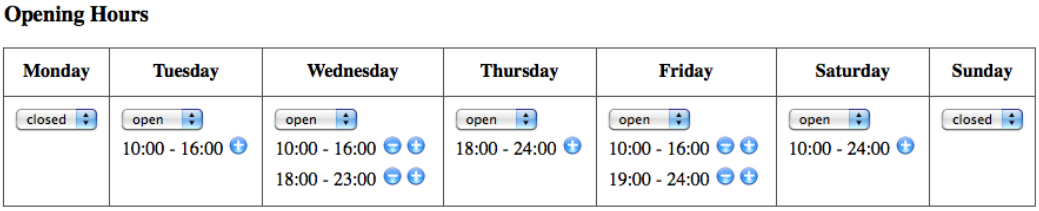

Figure 4. Editing a timetable

The Figure 5 shows a restaurant menu. As it is displayed, the user has already entered two courses (*Entrées* and *Plats*). In the first course, she has entered three dishes (*Salade de crudités*, *Salade de cabécou* and *Salade de coeurs de canard*), and she has entered various information about the price of dishes. This example shows that repeated components can be nested to create hierarchical structures (e.g. dishes within courses). As it will be explained later, the template author can control the insertion point of the *plus* and *minus* buttons inside the document, and their size. Also, some information, such as the specialty, some comments, or the prices are optional.

The bibliographic entry in Figure 6 is an extract of a bibliographic reference list. It shows that a bibliographic entry is made of two nested components. The first component sets the general category of the bibliographic entry, such as a *Paper*, while the second component sets a subcategory. In the figure, the choices for the second component are between *Article*, *ArticleInJournal*, *ArticleInProceedings* or *InBook*. These are all different types of bibliographic references that can describe a *Paper* publication. The presentation and the nature of the information displayed in the user interface may be quite different for each type.

Figure 5. Editing a menu

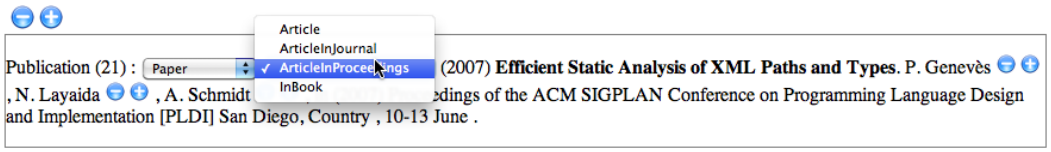

Figure 6. Editing a bibliography

The XML content below is generated from the document shown on Figure 6. This illustrates the kind of XML content that can be edited with AXEL and XTiger XML.

Example 3. XML content of a bibliography

```
<Publication>
  <PublicationId>21</PublicationId>
  <Paper>
    <ArticleInProceedings>
      <PublishingYear>2007</PublishingYear>
      <Title>Efficient Static Analysis of XML Paths and Types</Title>
      <Authors>
        <Author>
          <FirstName>P.</FirstName>
          <LastName>Genev&#xE8;s</LastName>
        </Author>
```

```
        ...
      </Authors>
      <Proceedings ShortName="PLDI">
        <PublishingYear>2007</PublishingYear>
        <ProceedingsTitle>Proceedings of the ACM SIGPLAN...
        </ProceedingsTitle>
        <Location>
          <City>San Diego</City>
          <State>California</State>
        </Location>
        <Dates>10-13 June</Dates>
      </Proceedings>
    </ArticleInProceedings>
  </Paper>
</Publication>
```

3.2. Constraining document structure

The examples above show the three types of structural constraints that define the directions into which the user can develop a document. These are: repetition, optionality, and choice.

Repetition is expressed as a `xt:repeat` element. It allows the user to insert its children elements between `minOccurs` and `maxOccurs` times. The example below shows how the *open* component of the opening hours table shown in Figure 4 defines a repetition of time slots which are themselves declared in a *slot* component. The generated XML content is visible on Example 4.

Example 4. Extract from the timetable template with repetition

```
<xt:component name="slot">
  <p>                    <!-- 7:00 opening hour hint -->
    <xt:use types="text" label="begin">7:00</xt:use>
    <span> - </span>     <!-- 24:00 closing hour hint -->
    <xt:use types="text" label="end">24:00</xt:use>
    <xt:menu-marker size="16"/>
  </p>
</xt:component>

<xt:component name="open">
 <br/>
 <xt:repeat minOccurs="1" maxOccurs="*" label="slots" >
   <xt:use types="slot" label="slot"/>
 </xt:repeat>
</xt:component>
```

Example 4 also shows the `xt:menu-marker` element that indicates where to insert the repetition buttons. It can have a `size` attribute to set the size of these buttons. In absence of the `xt:menu-marker` element, the buttons will be inserted after the last children of the `xt:repeat` fragment. You can notice that it can be declared inside the definition of a component type which is repeated, and not only directly inside the `xt:repeat` element.

Optionality is expressed as an `option` attribute which can be declared on the `xt:use` element inserting a component type in the document as shown on Example 5. When the value of `option` is `set`, the component target XML content is by default generated in the document but this can be changed by the user, by unchecking the checkbox which is displayed in the editor. When the value of `option` is `unset`, the component target XML content is not generated automatically, but this can be changed by the user, by checking the checkbox. When the attribute `option` is not defined, the XML content is always generated in the document. It is also possible to make parts of a document template optional using an `xt:repeat` element with `minOccurs` set to 0 and `maxOccurs` set to 1.

Example 5. Extract from the menu template with optionality

```
<p class="specialtyComment">
  <xt:use types="text" label="comment" option="unset">commentaire</xt:use>
</p>
```

Finally choice is expressed as an `xt:use` element where the `types` attributes declares several component types that can be inserted. The extract below shows that for a day of the week in the timetable template, in that case *Wednesday*, the user can select between two components, `open` or `closed`.

Example 6. Extract from the timetable template with choice (i.e. open vs. closed)

```
<xt:component name="open">
  ...see example above...
</xt:component>

<xt:component name="closed">
  <span/> <!-- this generates no XML content -->
</xt:component>

...
<td><xt:use label="wednesday" types="open closed"/></td>
...
```

3.3. Defining the XML content model

The template language drives the generation of a target XML content model with only two instructions: the `label` attribute and the `xt:attribute` primitive component type inclusion element.

The `label` attribute can be set on any `xt:use` component inclusion element, on any `xt:repeat` component repetition element, or on the `xt:head` element. Each `label` attribute generates a new XML element in the target content model during the serialization process. Similarly, when an `xt:use` element includes a choice of several component types, the selected component type generates a new XML element name after the component type name. The `label` attribute set on the `xt:head` element defines the target content model root node.

For example, the `<xt:use label="wednesday" types="open closed"/>` instruction in the opening hours table example generates a `<wednesday>` element that may contain either an `<open>` or `<closed>` child. If the user selects the *open* component type, this component generates a `<slots>` element (see the label of the `xt:repeat` in Example 4) that contains a repetition of `<slot>` elements with their own content model. This is illustrated below:

Example 7. XML content extract generated from the document on Figure 4

```
<wednesday>
  <open>
    <slots>
      <slot>
        <begin>10:00</begin>
        <end>16:00</end>
      </slot>
      <slot>
        <begin>18:00</begin>
        <end>23:00</end>
      </slot>
    </slots>
  </open>
</wednesday>
```

The `xt:attribute` element is a special component type inclusion element which can be used as an alternative to an `xt:use` element. The difference is that the target XML content model will be treated as an XML attribute instead of an element. This attribute, named after the *name* attribute of `xt:attribute`, will be attached to the current target XML element. As a consequence, the `xt:attribute` element can only include primitive component types that generate text data in the content model.

As an example, the `xt:attribute` element in the *price* component of the menu template visible on Figure 4 and listed below generates a `currency` attribute with

the possible values *EUR*, *CHF* or *USD*. In that particular case the choices in the popup menu are labelled from the content of the `i18n` attribute, here some currency symbols. It uses the primitive editor `select` which is described in Section 4.2.

Example 8. Price component type definition in the menu example

```
<xt:component name="price">
  <xt:menu-marker size="12"/>
  <xt:use types="text">prix</xt:use>
  <xt:attribute types="select" name="currency" values="EUR CHF USD"
                i18n="&#8364; CHF $" default="EUR"/>
</xt:component>
```

The *price* component of Example 8 can be inserted with `<xt:use types="price" label="price"/>`, which generates XML elements such as `<price currency="EUR">10</price>`.

4. Editing User Experience

AXEL employs two types of user interface controls: structure editing controls and primitive editor input fields that you can see in action in the examples of Section 3.1. The next sections explain these controls. The global editing user interface also supports tab navigation to jump from one primitive editor to the next (resp. previous) one. It is also possible to shift-click on a *minus* button to cut an item instead of deleting it, and to shift-click on a *plus* button to paste it at another position in the same repeat group.

4.1. Structure Editing Controls

Unselected document parts, either because they are repeated and their current count is 0, or because they are optional, are grayed out. We have implemented this behavior directly within the library with some CSS rules and some CSS generated `class` attributes.

The `xt:repeat` element generates a *minus* button and a *plus* button to respectively insert or delete a component. It can also replace the *minus* button with a checkbox if its `minOccurs` attribute value is 0: the checkbox is unchecked and the *plus* button is not visible if the user has not yet inserted a single item. The user can then insert a first item by checking the box. Then, if `maxOccurs` is greater than one, a *plus* button also appears to insert more items. The checkbox is replaced by a `minus` button as soon as the user has inserted more than one item, if this is allowed by `maxOccurs`.

The checkbox is also displayed when an `option` attribute is present on an `xt:use` element. In that case it is displayed as an isolated checkbox which is selected or not, depending on the existence of the component.

After some trials we finally decided to not give more feedback to help the user identify the boundaries of the components which are repeated or optional. Some subjective testing with automatically added borders were not satisfactory, mainly because they broke the document appearance. Our feeling is that in most cases the template content gives enough cues about the document internal structure. However, template authors can still create additional feedback, such as borders, using CSS and Javascript to achieve special effects on a per-template basis.

The `xt:use` element with multiple component types inclusion generates a pop-up menu that presents the different type names. These names can be internationalized by the individual component type definitions with an `i18n` attribute. Each time the user selects a different component type for inclusion, the currently visible component is replaced by the selected one. Actually the pop-up menu, an HTML `<select>` element, is inserted in place of the `xt:use` element, just before the included content.

4.2. Primitive Editor Input Fields

The library needs also to handle text user input so that users can create XML content, and not just an XML structure. This is done through primitive component types which are implemented by primitive editors. AXEL has a plug-in mechanism for creating new primitive editors by adding Javascript classes. Currently we use two of them: a `text` primitive editor, and a `select` primitive editor.

The `text` primitive editor manages a block of text which is displayed within a `<span>` XHTML element when not editing, and within an `<input>` or a `<textarea>` element when editing. It is possible to configure several parameters with the `params` attribute of the `xt:use` element that inserts a primitive editor. This attribute contains key-value definitions that influence the behavior of the editor. For instance the `layout` parameter can be set to *placed* so that the input field (i.e. `<input>` or `<textarea>`) dynamically replaces the `<span>`, or to *float* so that it is displayed over, with an initial equivalent shape. In all cases, these input fields grow as the user is typing, which gives a user experience similar to editing a WYSIWIG document such as in Word or Google Docs, and very different from editing a form, such as with XForms. Moreover, it is possible to configure them so that the input field inherits the typography from the document template, or to use `class` attributes to apply CSS effects to it.

We have extended the `text` primitive editor with the notion of *filters* that can be set on the editor. These filters, as the primitive editor, are implemented with Javascript classes following a specific API. Basically they can filter the target XML content to support complex XML structures which cannot be defined just with XTiger canonical constraints. They can also dynamically change the `<span>` static text display to any other XHTML presentation for the data. For instance, this article has been edited mainly with a `text` primitive editor that uses a `wiki` filter for creating

verbatim or emphasized text within paragraphs. This filter is shown in action on Figure 7. We have also managed for some configurations to position the cursor just after the character that was clicked to bring up the editing view. This creates a nice *click through* illusion that contributes to give a WYSIWIG feeling.

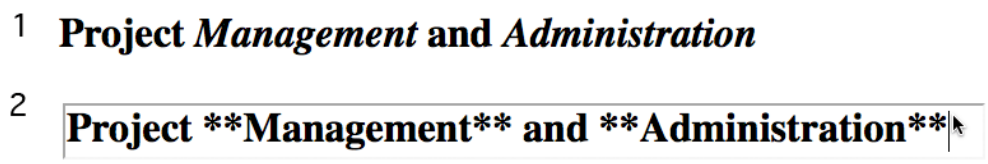

Figure 7. Text primitive editor with wiki filter while viewing (1) and editing (2)

The `select` primitive editor is rendered with a pop-up menu for directly selecting a value for an `xt:attribute` element as presented previously. It appears when the user clicks the current value of the field, which is displayed within a `<span>`, as with the `text` primitive editor. Figure 8 shows the popup menu after the user has clicked on the top *CHF* currency to change it.

Figure 8. Example of popup menu of the primitive editor "select"

Additional primitive editors will be created in the future to enhance the user experience, ranging from calendar selection controls – to pick up dates –, to file uploader editors – to insert images. This is fully supported by the XTiger XML concept of primitive component type.

4.3. Customization of Editing Behavior

The authors of template have the freedom to customize the editing user experience through CSS and Javascript programming. For this purpose, AXEL offers predefined CSS classes which are added to the generated user interface controls. For instance, we have experimented a very useful feature that consists in hiding all the structure editing controls by adding a *preview* class attribute to the `<body>` element of a template and some CSS rules. This gives an overview of the document, as in Figure 9.

The document of Figure 9 is still editable as all the primitive editors are functional, however the user cannot change the structure (adding or removing components in repetitions and changing the selected component of a `xt:use` choice). This can be useful to make light corrections (spell-checking, typos) to a document or to

translate it. This opens up future possibilities to generate different user interfaces for different tasks.

Figure 9. Menu document with all structure editing controls hidden but still editable

5. Server Integration

AXEL is designed to serve as the XML authoring layer for REST applications. It is fully implemented in Javascript; hence it can be executed in any browser providing reasonable support for the DOM model. This section outlines the main features of the library for that purpose.

5.1. Client-side Library

AXEL is a client-side Javascript library that takes as input an XHTML DOM tree containing the template. It transforms it into an XHTML tree containing the editing user interface and some Javascript objects for controlling editing. A programmer creates an editor by first instantiating an `xtiger.util.Form` object. The constructor parameter is a location path to a folder that contains the four images used by the editing user interface. Then s/he can set the template to transform with a call to `setTemplateSource` before calling the `transform` method to actually generated the editing user interface.

Example 9. Javascript code to transform a template loaded in the document object

```
var form = new xtiger.util.Form("{path-to-images}");
form.setTemplateSource(document);
form.transform();
```

By default the template is transformed in place. It is also possible to indicate a target XHTML document and a node inside which the result should be placed. The document that hosts the result of the transformation must include some specific AXEL CSS rules.

This very thin API is designed to embed XML authoring in any web application. For instance a programmer can build a generic XTiger XML editor that loads an XTiger XML template into an `iframe` element and transforms it to generate the editing user interface. It is even possible to directly put the code to generate the editor directly within a template file. This enables funny applications such as sending a template document by email to somebody, who can open it in his or her browser to make some editing, and then save the results to a server (this requires a little Javascript programming as explained in Section 5.2).

5.2. XML Serialization

The editor object has a function that serializes the current content of the document into the target XML content model. That function takes as input a logger object for accumulating the result while iterating on the document. A specific `xtiger.util.DOMLogger` is available for that purpose. It provides a method to dump its content into a string which can then be sent to the server as an HTTP PUT or POST request using the Ajax XHR object, now implemented in all major browsers.

Example 10. Javascript code to serialize the XML content of a document

```
var logger = new xtiger.util.DOMLogger ();
data = form.serializeData (logger); // form from above
var xmlString = logger.dump();
```

Reciprocally, the editor object provides a function that loads some XML data into a document. That function takes as input a DOM data source for iterating on the XML data while iterating on the template. A specific `xtiger.util.DOMDataSource` is available for that purpose. This object accepts an XML document object as input, such as the *responseXML* object constructed by an Ajax XHR object, or it can directly parse a string into an XML document object using a *DOMParser* object (or a simulated version if it is not available). This makes it quite easy to retrieve the XML data from a server side application and to inject it into the document. For experimental purposes we have also built a DOM data source based on the E4X Javascript API to natively parse XML data.

Example 11. Javascript code to load XML content from an Ajax XHR object into a document

```
var xhr = new XMLHttpRequest();
...
var source = new xtiger.util.DOMDataSource();
source.initFromDocument(xhr.responseXML);
form.loadData(source, result); // form from above
```

We currently have successfully integrated the library with WebDAV servers, with Orbeon Forms applications, and even with a Ruby on Rails application.

5.3. Experiments and Performances

The Javascript library has been developed and tested with Firefox (version 3 and above). Thanks to this client-side approach combined with the use of standard web technologies, we provide a platform-independent solution. It runs on recent versions of all major browsers although we did not do extensive testing on all platforms yet. The Javascript library compressed with the Yahoo UI compressor is less than 96 kB, including the basic editor plug-ins that were required to write this article.

We have experienced this Javascript library with several document templates in order to evaluate this solution in terms of performance and usability. The tested templates range from simple form-based structures, such as the menu example given previously, to more complex templates, like the template used for writing this paper. The table below lists some properties and some results obtained with different use cases of templates. Time results of these tests have been obtained on a MacBook Pro with 2.16 Ghz Intel Core Duo with 2GB memory running Firefox 3.5.3 (a 3 years+ computer).

Template Name	# nodes	Tree depth	Typical document size	Download time	Click response time ("-" means not noticeable)
Menu	89	5	2kB	-	-
Curriculum	173	4	2kB	-	-
Article	317	7	10-100kB	0.2-2s	-
Specification	243	6	50-500kB	1-36s	-

Figure 10. Loading XML documents into a template and editing with AXEL

We can notice that the download time is not noticeable for normal size XML documents. Download time depends on the complexity of the template characterized by its total number of node (*#nodes*) and the number of nested components (*tree depth*), but we haven't precisely characterized the relation yet. This is left as a future work in order to give guidelines to template authors. XML download time is linear

with the XML document size and, for instance, it took up to 36 seconds for a 500 kB XML document on this 3 years+ hardware configuration, which can be considered very long. However, the document was 190 printed pages in the browser and it was by itself an excellent surprise to be able to edit so long documents within the browser. This template also uses a specific plugin editor to edit logical expressions that take time to convert when loading and saving. The times given in the table do not take into account the preliminary download time of the template and the transformation time to turn the template into an editor, which are not noticeable in all the cases. The files were directly loaded from the local drive. We could also have presented the results with Safari which are even better.

We do not show a similar table for the time taken to generate XML content from the edited document because it is usually not noticeable, about 5 seconds in the worst case from the examples in the table above.

In all cases we noticed pretty good response time in the browser, between a click and the display of the input field. Usually not noticeable, it stays below half a second even with 500kB+ XML documents. These results obviously come from our client-side editing code, where only load and save operations pay for communication time. Thanks to recent web technologies, modern web browsers can become valuable XML document editors with the addition of a very tiny amount of code.

The various templates we have tested aim at evaluating both the expressive power of the XTiger XML template language and the quality of the authoring interface. The original XTiger language implemented in the Amaya stand-alone application has demonstrated its ability to define templates for editing a wide range of web documents [13], and due to its filiation with that language, XTiger XML inherits this feature. But with XTiger XML we gain in usability for simple applications, while keeping the expressive power for complex XML structures. Indeed, menu and curriculum vitae documents, for instance, can be straightforwardly edited by XML unaware people (for instance, a restaurant chef). For such "minimal" templates (few element types and little structural depth), the interface can be considered as minimal and allows users to focus on what they have to do (Hick's law [3]). Moreover, editing controls (menus for choices, add/suppression buttons for repetitions, text editing areas) are always contextually located and therefore mouse movements are kept minimal (Fitts's law [2]). We can notice that this simplicity at the authoring step does not prevent rich server-side services to be provided, such as databases feeding or high quality publication.

6. Related Works

By design, XTiger XML is an extension of the original XTiger template language. We tried to keep both languages as close as possible to each other, and indeed XTiger XML adds only a few elements and attributes. The main difference lies in the editors that are based on the languages. The original XTiger template language has been

implemented in the Amaya web editor, and as far as we know, it is available only in this editor. For XTiger XML, a very different approach was taken. It is implemented as a Javascript library, and then, it can run in any modern web browser.

As stated above, XTiger XML can be used as a form editing framework. It could even be possible to create plug-in modules to handle all the usual form interactors (drop lists, radio button, etc.), but the current syntax is not powerful enough to express non-structural constraints on the editable data model. However, this is not our main objective. Our priority is to develop further the library for editing document-oriented data and for managing collaborative editing constraints in the future.

The approach presented in this paper is different from wiki-based authoring environments, which are common on the web, such as MediaWiki in use by Wikipedia. First, it does not require the user to learn a specific syntax. Second, it is not limited to creating HTML content models. To some extent, semantic extensions of wikis such as Semantic MediaWiki make it possible to create annotated documents that contain structured information, but the requirement to learn a domain specific syntax still holds true. To cope with this issue, several template-based wikis have been proposed with more or less freedom for the author: the wiki template can be used simply as a seed document, without any control on the further authoring process, or it can on the contrary act as a very strict editing framework, preventing the author from deriving from the expected structure unless he modifies the template [10].

An in-between approach is proposed in [4] where a post-hoc validation mechanism can be applied on instances. While this can be satisfying for simple web pages, it becomes boring for the author when structures are more complex. Again, users have to choose between expressiveness and usability [5]. We claim that both can be provided thanks to the use of a rich template language and new editing paradigms such as those described in this paper. It can be noticed that a similar approach was recently proposed in [6] for the authoring of semantic annotations in MediaWiki, where editing controls are generated from ontologies.

XTiger XML with AXEL is very close in its objectives to generic XML authoring applications, especially those that use style sheets or XSLT transformations to associate a visual presentation with an XML content model defined by a schema. Such applications generate a WYSIWYG editor based on this visual presentation and driven by the schema. For a long time these tools were available only as desktop applications, such as XMLSpy or Oxygen. But with the success of web applications, browser-based WYSIWYG XML editors are appearing now, such as XOpus.

However, there are still two important differences:

- WYSIWYG XML editors, even when they run in the browser, require at least an XML schema and some XSLT transformations. This implies that they are used only for documents and data for which such resources exist, or that new schemas and transformations must be created for new types of documents and

data. We believe that the use of document templates instead of schemas and transformations languages makes XTiger XML easier to learn and to manipulate.

- XTiger XML and AXEL work on the web. They use the browser not only as the local platform for running the editor, but also as a web client that can download and upload documents on remote web servers. This allows everyone on the web to edit XML data and documents and to feed servers with well structured data. This also provides a global infrastructure for sharing and collaborating on XML contents.

7. Future Works

XTiger XML and AXEL are still under development, firstly to extend authoring services with new primitive editors. We have also identified some issues that will be addressed shortly.

7.1. Validation

In its current state, AXEL can be compared to XML tools based on XSLT, regarding validation: there is no formal way to make sure that the XML data generated by the tool are valid against a given schema, except by validating every instance. But there are solutions to this issue. XTiger can be seen as a regular tree grammar, and then an XTiger XML template can be compiled into a schema language (DTD, XML Schema, Relax-NG). But instead of producing schema language syntax equivalent to a template, our plan is to compile XTiger into the logical representation introduced in [8]. We can then use the validation services offered by the XML Resolving Solver [9], which is based on this logical representation. The Solver can statically check properties of XTiger XML templates, in the same way it checks properties of schemas.

A property that the Solver could statically check is whether a template always generates XML instances that are valid against a given schema. This may ensure for instance that the data entered with a template can be safely stored in an XML data base defined by its XML Schema.

7.2. Data Delivery

A second issue concerns data delivery on the web. Producing pure static XHTML read-only format remains of importance. This is required to enhance the probability to be indexed by major search engines. Explicitly including micro-formatted elements is another concern; for example, applications aiming at aggregating information (such as the Kritx review aggregator) rely on the availability of such information in XHTML documents. Those requirements are not met if documents are published only through a client-side Javascript transformation process.

We have manually transformed several document templates into an XSLT transformation that takes an XML content conform to the implicit document model as input, and generates an XHTML document that looks like the document generated from the document template for editing. We are confident that this process can itself be programmed as an XSLT transformation taking as input a XTiger XML template and generating the target XSLT transformation. Doing this will allow us to provide a full XML editorial chain within the browser.

7.3. Template Editing

The templates we have used up to now were created by various means. The simplest templates were written totally "by hand". In some cases we have used an XHTML editor (especially Amaya), for creating a skeleton document, that was then edited as a text file for adding the XTiger XML elements. Some other templates were almost entirely created with Amaya, which includes a feature for editing the original XTiger language. But, as stated earlier, there are differences between the XTiger language implemented in Amaya and the XTiger XML language. This implies that templates produced by Amaya have to be tweaked "by hand" when used by AXEL for editing XML documents. An obvious work item for the future is to adapt Amaya and implement all the extra XTiger features included in XTiger XML.

In addition to these means, we think that it would be interesting to also have a template authoring application running in the browser, to make that feature more widely available, without the need for installing a dedicated application. Such a tool could start from an initial XHTML document and allow the user to insert XTiger XML elements, in the same way as in Amaya. It would also be interesting to go a step further, when an XML Schema (or Relax-NG) is available for the target XML language. This schema could then be used to drive the interaction with the user in order to create a template consistent with this schema.

8. Conclusion

The XTiger XML language and its implementation in the AXEL library make it easy to create and update XML data and documents on the web. The language is very simple: it contains only a handful of elements and a few attributes. This allows anyone familiar with HTML to create templates. The editing engine provides a simple and intuitive user interface, which allows any web user to create and edit XML data.

Being implemented as a light Javascript library that runs in a web browser, the editor can be deployed very efficiently and it lets users free to work with their preferred tool. Because templates are based on XHTML, they allow XML data to be involved in any web application and to be collected by average web users. Building

on the ubiquitous web infrastructure, this approach helps to put XML within reach of everyone.

9. Acknowledgements

Early work on this project has been initiated in the framework of the PALETTE Integrated Project supported by the IST programme of the European Commission (DG Information Society and Media, no. 028038). We especially thank Thibaud Latour from CRP Henri Tudor, our partner in Luxembourg, for his active support. Further development of the XTiger XML language is currently supported by the Innovation Promotion Agency of Switzerland under the grant No 10813.1 PFES-ES, a project in collaboration with the MadeinLocal company (www.madeinlocal.com[1]).

Bibliography

[1] Amaya: Home page. http://www.w3.org/Amaya/

[2] P. M. Fitts: The information capacity of the human motor system in controlling the amplitude of movement. Journal of Experimental Psychology, Vol 47 pp. 381-391, 1954

[3] W. E. Hick: On the rate of gain of information. Quarterly Journal of Experimental Psychology, Vol 4 pp. 11-26, 1952

[4] A. Di Iorio – F. Vitali: Wiki content templating. WWW 2008, Beijing, China, pp. 615-624, ACM Press, 2008

[5] A. Di Iorio – S. Zacchiroli: Constrained wiki: an oxymoron?. WikiSym '06: Proceedings of the 2006 international symposium on Wikis, pp. 89–98, ACM Press, 2006

[6] A. Di Iorio – S. Duca – S. Righini – D. Rossi – F. Vitali: Customized Edit Interfaces for Wikis via Semantic Annotations. Workshop on Adaptation and Personalization for Web 2.0, UMAP'09, June 22-26, 2009

[7] F. Flores – V. Quint – I. Vatton: Templates, Microformats and Structured Editing. Proceedings of the 2006 ACM Symposium on Document Engineering, DocEng 2006, pp. 188-197, ACM Press, October 2006

[8] P. Genevès – N. Layaïda – A. Schmitt: Efficient static analysis of XML paths and types. PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 342–351, ACM Press, 2007

[1] http://www.madeinlocal.com

[9] P. Genevès – N. Layaïda: XML Reasoning Made Practical. Proceedings of the 26th IEEE International Conference on Data Engineering, ICDE 2010, IEEE, March 2010. http://wam.inrialpes.fr/publications/2010/ICDE10demo.pdf

[10] A. Haake – S. Lukosch – T. Schummer: Wiki-templates, adding structure support to wikis on demand. WikiSym' 05, San Diego, USA, pp. 41 – 51, ACM Press, 2005

[11] É. Kia – V. Quint – I. Vatton: XTiger Language Specification. http://www.w3.org/Amaya/Templates/XTiger-spec.html

[12] P. Nálevka: Advanced Automated Authoring with XML. XML Prague 2009

[13] V. Quint – I. Vatton: Structured Templates for Authoring Semantically Rich Documents. Proceedings of the 2007 international workshop on Semantically aware document processing and indexing, ACM International Conference Proceeding Series; Vol. 259, pp. 41-48, ACM, 2007

[14] S. Sire: XTiger XML Language Specification. http://media.epfl.ch/Templates/XTiger-XML-spec.html