Techniques for Authoring Complex XML Documents

Vincent Quint, Irène Vatton

Techniques for Authoring Complex XML Documents

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

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

ABSTRACT

This paper reviews the main innovations of XML and considers their impact on the editing techniques for structured documents. Namespaces open the way to compound documents; well-formedness brings more freedom in the editing task; CSS allows style to be associated easily with structured documents. In addition to these innovative features, the wide deployment of XML introduces structured documents in many new applications, including applications where text is not the dominant content type. In languages such as SVG or SMIL, for instance, XML is used to represent vector graphics or multimedia presentations.

This is a challenging situation for authoring tools. Traditional methods for editing structured documents are not sufficient to address the new requirements. New techniques must be developed or adapted to allow more users to efficiently create advanced XML documents. These techniques include multiple views, semantic-driven editing, direct manipulation, concurrent manipulation of style and structure, and integrated multi-language editing. They have been implemented and experimented in the Amaya editor and in some other tools.

Categories and Subject Descriptors

H.5 [**Information Interfaces and Presentation**]: User Interfaces—*Graphical user interfaces, Interaction styles*; I.7 [**Document and Text Processing**]: Document Preparation—*Languages and systems, Markup languages, Standards, XML*

General Terms

Design, Experimentation, Languages

Keywords

XML, authoring tools, structured editing, direct manipulation, compound documents, style languages, CSS

DocEng'04, October 28–30, 2004, Milwaukee, Wisconsin, USA.

1. INTRODUCTION

Authoring techniques for structured documents constituted an active research area during the second half of the 80's and the early 90's [10]. Several experimental systems such as Grif [7] and Rita [6] were developed and a few production tools resulted from that work. Most of them, such as Author/Editor and ArborText, were SGML oriented. Since that time, the specific requirements of the Web led W3C to create XML on the basis of SGML. This new language contributed strongly to popularize structured documents. New tools emerged for authoring XML documents, but most of them were straightforwardly derived from the previous SGML tools and from their underlying concepts, implementing only the XML features that were already available with SGML.

The benefits of XML do not consist only in a simplification of SGML. They also stem from the new concepts and languages that were developed specifically with XML. Using only the SGML techniques to manipulate XML documents is missing a significant part of the potential advantages from XML.

In this paper, we identify the major advances in the authoring techniques for structured documents that are made possible by the XML technology and we illustrate them with features we have implemented in Amaya and in some other editors. Amaya [1] is an authoring tool for the Web that integrates seamlessly editing, browsing and publishing features. It focuses on the standards developed by IETF and W3C to produce valid documents. It is used for maintaining Web sites where conformance to standards is considered important. It supports several document formats: generic XML and such XML applications as XHTML for hypertext, MathML for mathematical expressions, and SVG for vector graphics (with SMIL animation). It also supports the CSS style sheet language.

After this brief introduction, the next section gives an overview of the approaches that are usually taken in different editors used to create and edit XML documents. Section 3 reviews the main features offered by XML that differentiates it from its predecessor, SGML. Section 4 constitutes the main contribution of the paper. It considers the impact of the new features on authoring tools and proposes some innovative techniques for handling complex XML documents in an authoring environment. These techniques are presented in terms of benefits for the user and requirements for the software engineer. Finally, a conclusion summarizes the contributions of the paper.

2. USUAL APPROACHES

Many popular XML editors are basically text editors that are aware of the XML syntax and follow a given DTD or XML schema. The knowledge of the syntax allows the editor to help users in the syntactic details and ensures that a well-formed document is produced. The DTD or schema is used to check the structure of the edited document and thus allows the tool to assist the user in producing a valid document [13] [12]. Usually these tools may follow any DTD or schema. Formatting, although sometimes possible, is not the main concern for interaction, the tools focusing often on syntax and structure. In that perspective, users can usually see at least two views of their documents, the Source view showing the XML code and the Structure view displaying the DOM tree. A third view is often provided, to show a formated representation of the document, but in many cases, this view is just a way to preview the document layout and does not allow the user to really edit.

As opposed to these "generic" tools that can work with any document type, there are also specialized tools that are dedicated to a single document type, most often HTML or XHTML, such as Mozilla Composer or DreamWeaver. The purpose of these tools is to hide the complexity of structured documents, by reproducing as much as possible the behaviour of word processors. They allow naive users to create and modify a complex document by manipulating a formatted representation of the document in a view called the Normal view. They require little knowledge from the user about structured documents and markup languages. The tool itself, being tailored for a specific document type, does not really need a DTD or a schema, as the knowledge contained in a DTD is already captured in the code of the tool.

Obviously, the situation is not completely black and white. There are "generic" tools, such as XMetaL or XMLspy [14], for instance, that behave like specialized tools for some parts of structured documents. Tables are the typical case, provided they follow a well-known model, such as the HTML or CALS models. If it is the case, the editor provides easy to use, specialized commands for manipulating table cells, column and rows. But this ease of use is only available for a very limited set of sub-languages.

The ideal tool should combine the advantages of both categories. It should be flexible enough to support any document type and XML language, but it should also provide the kind of user-friendly interface that makes specialized tools easy to use.

3. NOVEL FEATURES IN XML

The main motivation of W3C for designing XML [5] was to enable the exchange of a wide variety of documents and data on the Web in a flexible text format compatible with SGML. The scope of SGML was broadened to encompass new types of documents, specifically *Web documents*. These documents typically integrate various distributed resources addressed by URIs (Uniform Resource Identifiers) and are related to other documents through hypertext links. Distribution and hypertext make a significant change as compared to many traditional SGML applications.

Another important change is the introduction of namespaces [4], which opens the door to compound documents. Namespaces allow an XML document to mix several markup languages that represent different parts of the document. These pieces can be nested within each other and grains of information from different markup languages can be freely combined, whatever their size. This offers structured documents a modular approach. Specialized markup languages can be developed for modelling very precisely each part of a document specifically, and these specialized languages can be shared and reused in different contexts.

This feature allows XML to cover a wide range of applications. A number of XML languages have been developed for very different types of structured data and documents. In particular, structured text, which was the main application area of SGML, represents only a fraction of the numerous XML applications available now. XML is used for encoding *non-textual information* such as vector graphics, mathematical expressions, synchronized multimedia documents, complex forms, etc., to mention only a few document-oriented applications. A broad range of components are thus available and they can be mixed for representing *compound structured documents*.

The combination of the Web dimension of XML and its namespace mechanism offers a number of new possibilities for complex, distributed documents, where various pieces in various markup languages can be included within each other, in a single file or by aggregating multiple distributed resources included inside each other.

SGML puts the emphasis on validity: every document must strictly comply with a DTD, which is compulsory for every kind of processing. XML relaxes this constraint by introducing the notion of well-formedness, which provides some *flexibility* in the processing of structured documents. This is a significant change, as documents that are supposed to comply with a DTD can now be processed even when the DTD is not available. Documents can also exist independently of any DTD or schema.

Style is the last important change we consider here, from the editing perspective. Style was an important issue for SGML documents, but there were a number of solutions and even the standard ones were not widely deployed. In fact, style was an unsolved issue. With XML, only two style languages are widely accepted and deployed, CSS (Cascading Style Sheets) [3] and XSL (Extensible Stylesheet Language), with separate fields of application but a lot of commonality (style properties, for instance are the same in both languages). Style plays an important role in documents, and that is probably one of the key differences with semi-structured data. Focusing this article on documents, we pay special attention to style and its role in editing XML documents.

4. EDITING XML DOCUMENTS

With all these new features, traditional methods for editing structured documents have to be revisited. This section analyzes the impact of these features on editing techniques and proposes some new solutions that were validated in Amaya. It focuses on Web documents, non-textual information, flexible structure manipulations, compound structured documents, and style.

4.1 Editing Web documents

As stated above, XML was designed for the Web. An XML authoring tool should therefore be somehow related with the Web. Given the key role of URIs in the Web archi-

tecture [8] and in XML technologies, an XML tool should at least handle URIs for what they really are, i.e. resource identifiers, and not as meaningless character strings. It should also offer means to handle hypertext links, both when they relate different parts of the same document and when they cross document boundaries to point to external documents or resources. Links are part of many XML languages, such as XHTML, SVG, and SMIL, not to mention XLink, an XML language dedicated to hypertext links. For that reason, links also require some dedicated features in an authoring environment.

To fill this requirement, Web access is needed. Being able to access the Web and to navigate, an authoring environment can help the user locate a given resource. The user can then check that the resource exists and make sure it is really what s/he meant. If the tool itself is handling the resource, it can manage its URI, relieving the user from the burden or typing or copying the URI, thus avoiding many errors.

We have developed Amaya following this approach. This Web tool integrates seamlessly a browsing functionality with the authoring features. It allows the author to browse the Web as well as local files and to display the resources of interest on the screen. In addition, when an element in an XML document has an `xml:id` attribute, it is considered as the potential target of hypertext links, and on user's request, it is displayed with a target icon, that can be clicked to set the end of a link.

Then, entering an URI or creating a link in the XML document being edited is done simply by clicking the corresponding resource or element on the screen. As the tool has itself downloaded the resource, it knows its URI and can copy it safely and automatically for the user. Moreover, the document being edited can be used to navigate the Web, if it contains some links or embedded resources. With this mechanism, documents are edited in their actual context and users are confident that the linked resources are the right ones.

The editor can directly save a document whatever its location thanks to the connection to the Web. From the user's viewpoint, there is no difference between local and remote documents: all documents are saved with the same command. Depending on its URI, the document is written on the local disk or through the Web onto a remote server. All resources are then seen in a homogeneous space that can be accessed in read and write mode transparently. XML documents are treated as Web resources.

4.2 Non-textual information

For many XML applications, editing a document through its source code is not appropriate. Source editing may work with textual documents, but with many of the new XML applications, this is simply not an option. One can not practically edit SVG graphics made of many shapes, just by editing the coordinates of their control points between pointy brackets. One can not safely manipulate the time dependencies of a SMIL multimedia document by editing the source of a structure combining `par` and `seq` temporal operators as well as their attributes.

The idea here is to allow users to edit documents according to the semantics of the XML languages involved. The semantics of SVG, for instance, is about 2D graphics. Users should be able to interact graphically with the geometric

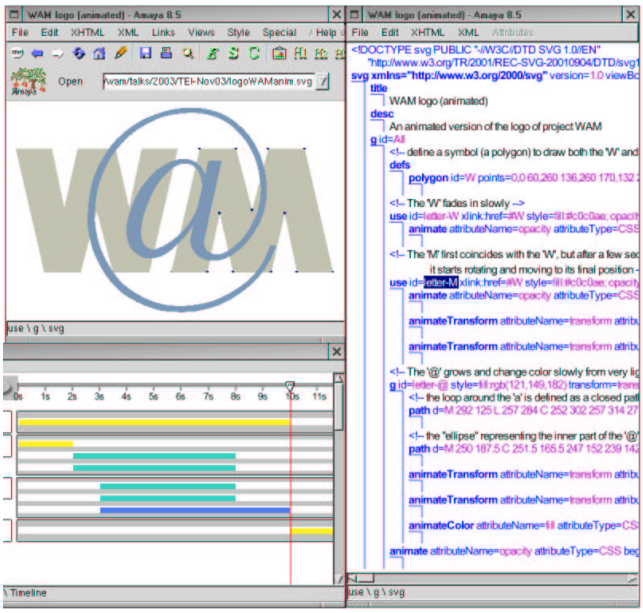

Figure 1: An animated SVG document in Amaya

shapes themselves, as opposed to their textual representation in XML. In the same way, the time structure of a SMIL document should be presented to users in a self-explanatory form that clearly shows how the time periods of all media objects are related with each other. Users should be able to manipulate these time periods in a natural way. Again, the idea is to handle the elements of an XML structure according to their semantics, regardless of the underlying XML structure or syntax.

In the area of human-computer interaction, this is the usual approach of direct manipulation [11], where users feel that they are directly controlling the objects of the application, without any intermediary. This approach has been taken in Amaya. SVG graphics are manipulated as graphical objects, with the same kind of interface as in drawing programs. Figure 1 shows a simple animated SVG document which contains two instances (`use` elements) of the same `polygon` that represent letters W and M, plus a group (`g` element) containing two closed `path`s that represents letter @. Three views are open. The top left window shows the Normal or Formatted view, where direct manipulation of graphics is possible. The control points of letter M are highlighted. They allow the user to use the mouse for moving or changing the shape (defined by attribute `points`) of the polygon.

Likewise, the time dimension of SMIL can be manipulated through a timeline, where each object with a temporal behaviour is represented by a bar whose length and position reflect the duration and start time of the object. The length and position of these bars can be modified with the mouse to change the synchronization of the corresponding object.

Actually, the timeline representation has been validated in two different authoring tools, Amaya and LimSee. In Amaya it represents the time dimension of all animated SVG elements. Thanks to the modularity of XML, the SVG language uses the animation module from the SMIL language. Document instances can then use SMIL `animate` elements within SVG elements to make them alive (see the Struc-

ture view which shows the DOM tree in the window on the right of Figure 1). The Timeline view (Figure 1, bottom left window) collects all `animate` elements in a document and displays them with different colors representing the property they animate (color, geometric transformation, motion, other).

LimSee [9] is an authoring environment dedicated to SMIL documents. It also provides a timeline, but this view is built in a different way. As time information in a SMIL document is hierarchically organized and constitutes the body of a document, the tree of the `seq` and `par` elements is also represented on the timeline in LimSee. In both tools, the user can move and resize the time bar to update such temporal attributes as `begin`, `end`, `dur`ation, etc. The user can also move a slider to observe the document at a given date (Figure 1 shows the document at 10s. from its beginning). This kind of manipulation has proven very useful, if not necessary, to manipulate efficiently the complex structures representing time in multimedia XML documents.

4.3 Flexible structure manipulation

Even with textual documents, high level manipulations make sense. Most XML editors offer context-sensitive menus or palettes that propose to the user all the elements that can be inserted before or after the current element. Although these menus are generally helpful, they are often considered by users as very constraining, while in many cases more flexible and efficient interaction can be provided. We have experimented with two different techniques that can be used in addition to the usual contextual menus: the Enter key, and automatic transformations.

4.3.1 Using the Enter key

In a plain text editor the Enter key is the main tool for structuring a document into lines and paragraphs. Pressing this key once breaks an existing line or creates a new line when the current position is at the end of a line. Pressing the Enter key multiple times introduces empty lines which indicate more important breaks in the sequence of text. We propose a mechanism based on this principle to structure XML documents with textual contents: the Enter key is used to break elements or to create new elements in the structure of an XML document.

The Enter key acts on the current selection. The current selection may be either a position in the document (a caret), with no content, or some structural element.

1. If a structural element is selected, the Enter key creates a new empty element of the same type after the selected element, and the selection, which becomes a position, is put within the new empty element.

2. If the selection is a position within some character string, the Enter key simply splits the current element at that position, replacing it by two elements, and putting the current position at the beginning of the second element. The choice of the element to be broken is based on the DTD: it is the first ancestor that accepts a following sibling; the type of the second element is also decided according to the DTD.

3. If the selection is a position at the beginning or at the end of a non empty element, a new, empty element is created before or after that element, respectively. The type of the empty element to be created is defined by the DTD, as well as its level in the structure. It is a sibling of the first ancestor of the current position that accepts a sibling. The current position is then put in the new empty element.

4. If the selection is a position within an empty element (for instance a caret in an empty XHTML `p`aragraph that is a child of a list item `li`), the Enter key first deletes this element (the paragraph). If this is not the first nor the last child of its parent (if it is a paragraph in the the middle of the list item), the first ancestor element that can be repeated according to the DTD is split at that position (the list item is divided into two list items). Then, in all cases, a new empty element, whose type is chosen according to the DTD (an empty list item), is created after the first part (the first list item resulting from the split) if there was a split, after the first repeatable ancestor otherwise. Finally, the current position is put in the new empty element (list item).

To continue the analogy with plain text editors, the action of the Backspace key is the opposite of the action of the Enter key, and is not detailed here. While the Enter key can split elements, the Backspace key can merge elements.

As an example, suppose we have an XHTML `ol` (ordered list) element containing three `li` (list item) elements. Suppose the current selection (the caret, represented by a vertical bar '`|`') is inside the second list item, at the beginning of the second sentence of this item. The internal structure of the document is as follows:

```
<ol>
  <li>Aaaaa</li>
  <li>Bbbbb. |Cccccc</li>
  <li>Dddddd</li>
</ol>
```

Pressing Enter splits the content of the item, creating within the item a new `p`aragraph which contains the second sentence (case 2):

```
<ol>
  <li>Aaaaa</li>
  <li>Bbbbb.
     <p>|Cccccc</p></li>
  <li>Dddddd</li>
</ol>
```

Pressing Enter a second time creates a new empty `p`aragraph between the two parts of the list item (case 3):

```
<ol>
  <li>Aaaaa</li>
  <li>Bbbbb.
     <p>|</p>
     <p>Cccccc</p></li>
  <li>Dddddd</li>
</ol>
```

If this is what you want, just go ahead and type the content of this paragraph. Otherwise, press Enter a third time. The empty `p`aragraph is deleted, the list item is split into two list items and a new, empty list item is created in between (case 4):

```
<ol>
  <li>Aaaaa</li>
  <li>Bbbbb.
  <li>|</li>
  <li>Cccccc</li>
  <li>Dddddd</li>
</ol>
```

If this is what you want, type the content of this new list item. Otherwise, press Enter a fourth time: the list is split into two lists and a new empty paragraph is created in between (case 4 again):

```
<ol>
  <li>Aaaaa</li>
  <li>Bbbbb.</li>
</ol>
<p>|</p>
<ol>
  <li>Cccccc</li>
  <li>Dddddd</li>
</ol>
```

We have validated this principle in Amaya. The experience has shown that some complementary features may greatly improve efficiency. The first improvement is achieved by providing the algorithm summarized above with a bit more information. In some cases, the DTD allows different types of elements to be inserted in the document structure at a given position. A choice has to be made by the user. A pop-up menu proposing all the possible types is not a solution, as the main idea is to allow structure manipulations to be done quickly, in a few keystrokes. A configuration file associated with the DTD is used instead. It indicates what type of element should be created by default when several options are available. As an example, for the XHTML DTD, this file says that a paragraph (`p` element in XHTML) should be created rather than a heading (`h1`, `h2`, `h3`, etc.) or a list (`ol`, `ul`, `dl`), when there is a choice between all these elements. The other elements, which are less frequently used than paragraphs, can be created too (see section 4.3.2), but they require a few more keystrokes. In Amaya, the configuration file is created manually. It could also be built automatically by counting the number of occurrences of elements in a representative corpus or be counting the number of elements created by a user.

Another important feature needed to make the Enter key effective is structure selection. A proper selection has to be made to take advantage of the Enter key. Selecting a position is done in the traditional way, either by clicking the position on the formatted document or by moving the caret (which represents the current position) with the usual arrow keys. Selecting a structural element is made easy by the Esc (or F2) key which selects the parent element of the current selection in the document structure. Thus, the user moves the caret to the element of interest, and by hitting this key several times, s/he can travel the tree structure towards the root, stopping when the desired element is reached.

This principle has proven very efficient for editing structured text, while preserving the validity of the edited document. Even if the algorithm may seem a bit complicated, doing different things in different contexts, its behaviour on most textual document types looks very natural. In addition, users can manipulate the document structure quickly, directly from the keyboard, with very few keystrokes. The algorithm works well for various types of documents, and requires very little adaptation for each type of document. Only the simple configuration file mentioned above has to be provided in addition to the DTD. This algorithm has also been successfully used for structures like mathematical expressions or tables. For instance, in MathML, a fraction can be split into two fractions very simply, or in XHTML tables, columns and rows can be added as simply as paragraphs or list items. As a consequence, the user interface is homogeneous. The same commands can be used to perform the same kind of operation on very different objects, and these commands are just an extension of a well known command for text files. The user has only to learn how to use a few powerful commands that work the same in many situations.

However, it should be recognized that not everything can be done that way. For instance, this editing mode does not help much when manipulating structured graphics in SVG. That is the reason why other editing modes are necessary.

4.3.2 Structure transformations

Structure transformations constitute a powerful way to manipulate XML documents, especially in an authoring tool, where users frequently change the structure of their documents. Structure transformations are involved in the editing commands discussed above: splitting and merging elements are transformations. From the user's perspective however, transformations discussed in this section are different, because they require the user to actually choose the element types, while it is the system that decides when using the Enter key.

It should be possible for the author to set the selection (a position, an element or a character string) and then ask the system to create an element of a given type there, simply by choosing the type in a menu or with a keyboard shortcut. Like the Enter key, the choice of an element type may lead to different actions on the structure, depending on the current selection:

1. if the selection is a position in an empty element, an element of the chosen type is created as a child of the empty element, if this is allowed by the DTD. If it is not allowed, the editor tries to create a series of descendents from the existing element to an element of the chosen type, again based on the DTD (in XHTML, an example is the creation of a `dt` element when the selection is in an empty `div` element: a `dl` is created in the hierarchy between the requested `dt` and the existing `div`). If this fails too, the editor tries to replace the empty element by an empty element of the chosen type.

2. if the selection is a position in a character string, the desired element is simply created at this position, if it can be inserted there legally. Otherwise the element(s) containing the current position are split, like in case 2 of the Enter key, and the desired element is created between the two parts.

3. if the selection is some character string, the new element replaces the character string in the structure, and the character string becomes its contents. The DTD may forbid this change. In that case, the ancestor elements are split until the new element can be

inserted between the two parts of the split element. An example of this in XHTML is when a character string is selected in the middle of a paragraph (`p` element) and the user wants to create a `h2`, which is not allowed as a child of a `p`. The paragraph is then divided into two paragraphs and a `h2` containing the selected text is created between the two paragraphs. The same principle applies when a sequence of character strings and elements is selected.

4. if a structural element is selected, or a sequence of structural elements, their types are simply changed into the desired type. In XHTML, a `p` element can be turned into a `h2` element, for instance. If this is not allowed by the DTD, an element of the desired type is created, wrapping the selected element(s). In XHTML, a sequence of a heading (`h1`, `h2`, `h3`, etc,) and some paragraphs (`p` elements) and lists (`ul`, `ol`, `dl`) can thus be grouped into a division (`div`). If this fails, more complex transformations have to be made.

Although very different structure manipulations are involved behind the scene, all these actions are seen by the user as the same command: create this type of element here, with this contents.

This command has been successfully implemented and validated in Amaya. To make case 4 useful, a lightweight transformation language [2] is used to handle more complex transformations. It is very close to XSLT, regarding its functionality, but with a more compact syntax. XSLT would certainly do the job, but this language was developed before XSLT was available and it was never replaced. A transformation sheet has to be written to specify all the transformations a user may need when manipulating structures in a given DTD. The editor looks in that sheet and executes the first transformation that matches the current selection and that generates the desired type. As an example, in XHTML, one can transform a sequence of paragraphs (`p` elements) into a list (`ul` or `ol`) of items (`li`), or conversely.

This command is considered very convenient by users. It solves indeed a significant problem of structured editors. The user no longer feels constrained by a tool that offers only contextual menus with only a few elements types for creating new elements. On the contrary, with this method, any element type defined by the DTD may be chosen at any time, and the tool makes its best efforts to create the desired element for the user.

Initially conceived for editing textual structures, this command demonstrated some advantages for editing other types of structures. In Amaya, it is extensively used for editing mathematical expressions in MathML. It allows any construct to be freely inserted anywhere, and many structures to be transformed very efficiently.

Another important advantage of this approach, is that it does not require the structure of an XML document to be built top down. One can start by typing plain text that is simply structured as a sequence of paragraphs, using only the Enter key as a structuring command. A hierarchical structure can be added afterwards. One can change the type of some paragraphs, to make section headings for instance. One can transform some sequences of paragraphs into lists. These elements can then be grouped into divisions, and so on, just by selecting the concerned elements and choosing an element type from a menu. This simple approach is very frequently used for creating and editing XHTML documents, for instance.

It is worth noting that, while it is very flexible, this process does not break the validity of the document. All basic operations involved in the process (the four operations listed above) are controlled by the DTD and always produce a valid structure. If the DTD does not allow one transformation to be performed, the command is simply aborted. Therefore, any sequence of these operations preserves the document validity.

4.4 Editing compound structured documents

In the previous section, we have seen how to manipulate different kinds of XML structures, but these structures are considered in isolation. However, when manipulating real documents, structured objects are usually embedded in a larger context: mathematical notations appear inside paragraphs; drawings are often part of some larger document, as well as tables or forms. So, to fully address the real issue we have to consider editing compound structured documents, i.e. the type of documents made possible by XML namespaces.

4.4.1 Namespaces and transclusion

There are two different ways to make compound documents. One is to build "monolithic" files where pieces in different markup languages are tightly integrated thanks to namespaces. This is for instance well suited for a scientific or technical document that includes a number of mathematical expressions and drawings. Another way to make compound documents is to take advantage of the hypertext nature of the Web, and to separate various pieces of a document in external resources. A typical example is a document that integrates a few drawings that are also used in some other documents. The external resources can thus be shared. Depending of the application, one approach or the other can be used. Moreover, both can be used simultaneously: equations may be part of the document itself, to avoid the burden of managing a large number of small files, while some complex animated graphics may be left as a separate resource and reused by other documents.

To allow users to take advantage of both solutions, an authoring environment should be able to handle a monolithic structure and a transclusion mechanism as well. Transclusion is actually part of many XML languages and should then be supported anyway to handle these languages properly. XHTML, for instance, uses the `object` element to include any external Web resource, or the `img` element to embed various types of images, including images in XML formats such as SVG. SVG, in turn, can include other resources by using the `foreignObject` element.

We have experimented both solutions in Amaya. On the one hand, the internal representation of XML documents, a DOM tree, has been implemented in such a way it can accommodate nodes from different namespaces in the same tree. On the other hand, a transclusion mechanism implements such "embedding" elements as `object`, `img`, or `foreignObject`, but transclusion is only used for display purpose. It has been considered too much confusing to allow the user to edit embedded elements in their embedding context. The tree representing the external resource is inserted in the DOM tree as a child of the embedding object, but it can not be edited, although the user can ask the ex-

ternal resource to be opened and edited as a separate document (Amaya can edit several documents simultaneously). Doing so, users can see the various resources in the way they are supposed to be seen, but they edit them as separate resources, what they actually are. This also avoids any ambiguity when saving a document. If several resources are embedded in the saved document, which ones should be saved, and where?

4.4.2 Views

Structured document editors usually propose several views of a document, displaying the same structure from different perspectives. We have seen in section 4.2 that, to offer direct manipulation, additional, specialized views are required, where structured objects are presented and can be manipulated according to their own semantics. As each of these specialized views is strongly dependent on the application semantics, several such views are needed when multiple XML languages are involved in a single document.

The usual Normal, Structure and Source views are generic; they apply to any XML application. They make sense for all parts of a compound XML document. However, each specialized view is meaningful only for the parts belonging to a given markup language. While the ones can display the whole document, the others have to show only some objects included in the document and are not needed when a document does not contain objects of the corresponding language.

The concept of multiple views is implemented in Amaya, with the generic and specialized views. The Timeline view mentioned in section 4.2 is a typical example of a specialized view that is useful only for some objects, animated SVG graphics. By displaying all `animate` elements contained in the document, it gives an overall idea of how the document is animated, even when animated graphics are interspersed in various parts of the same document.

Another specialized view is the ToC view for XHTML. It displays all headings (`h1`, `h2`, `h3`...) from the XHTML namespace that appear in a document, thus showing a table of contents. Here again, the idea is to give an overview of the document, by displaying all its section headings. This view is also used to move quickly across the document, thanks to the "synchronization" of all views: when clicking the representation of an element in a view, the document is automatically scrolled in all other views, to show the representation of the same element.

When using several XML languages, a number of views may be necessary to show and edit all aspects of a compound document, and this could become confusing. To avoid such a confusion, Amaya allows users to open and close views at any time. Amaya can also merge several views and display them in a single window. As an example, a document that combines structured text in XHTML, vector graphics in SVG and mathematics in MathML, may be manipulated in a single window, the Formatted (or Normal) window, which shows formatted text, real graphics and formatted math, thus providing the user with a natural and readable representation of the document. In this window, text can be manipulated as explained in section 4.3, graphics can be edited graphically, and math can be written in an efficient way too. Additional views may be open only when necessary, to see some particular aspects of the document or to perform some specific operations. A Timeline view is open to animate graphics, a ToC view to move quickly through the document, a Structure view to control the embedding elements, etc.

Specialized views play an important role in the understanding of the actual document structure. To reinforce that feature, they show in the same way elements that are part of a monolithic document and elements that are embedded by transclusion.

4.4.3 Editing modes

For complex compound documents, several editing modes have to be provided by the authoring environment, to allow users to work efficiently on all parts of a document. Multiple editing modes are required not only to face complexity, but also to take advantage of all aspects of the XML language. To identify the needed editing modes, it is worth considering the different levels of the XML language itself.

At the lowest level, an XML file is simply a flow of characters with a syntax that represents basically elements, attributes, and contents. At the next level, this syntax describes a tree structure: elements are nested, they have attributes, and contents constitute the terminal elements of the tree. This corresponds to well-formed documents. When a DTD or a schema enters into play, the tree structure is not only constrained by the rules of well-formedness, but also by the DTD or schema. This is the third level. The fourth and last level is the semantics level. Elements and attributes not only have a name and a position in a tree structure as indicated by the DTD or schema, but they also have a meaning and play a role in the document. This is not formally described, but usually explained in the documentation of the document type. Application semantics add more constraints on the way to combine elements and attributes in a meaningful structure.

When editing a document, it may be useful to consider it at these different levels of abstraction, and a specific editing mode could be associated with each level to perform different kinds of operation:

1. source code editing, to manipulate an XML file as plain text,
2. free structure editing, to manipulate the document as an unconstrained tree,
3. schema-driven editing, to manipulate the document tree according to the DTD or schema,
4. semantics-driven editing, to take into account the specific aspects of the XML application.

All these editing modes have been implemented in Amaya. To make them effective, they are all available simultaneously and the user is free to choose the commands of any mode at any time, without any explicit mode switch.

In the first editing mode, the user has to know the XML syntax, the structural constraints of the various XML languages, as well as their semantics. In addition to text editing, the editor provides syntax and structure checking on user's request. This mode is used by expert users when other modes are not suited to the task at hand. The Source view is well suited to this editing mode. In this view the XML syntax is highlighted for easier editing, and the user can make any change freely. At any time, a special command may be called that parses the modified source file, rebuilds

the DOM tree, reformats the document and redisplays it in all open views. If there is an error, the parser stops immediately, as required by the XML specification, pointing at the error in the Source view.

In free structure editing mode, the XML syntax is handled by the tool, but the user still has to know the structural constraints and the semantics of the markup language. Amaya does not need or use any DTD or schema in this mode. Formatting, performed by the tool on the basis of CSS style sheets, helps the user to visually check the document structure through formatting semantics. The view of choice for this mode is the Structure view, which displays a DOM tree. Every change done in this view is immediately reflected in the other views, and especially in the Formatted view. The Normal (or Formatted) view can also be used, as well as the structure manipulation commands presented in section 4.3, but validity is not checked, as there is no DTD.

In the schema-driven editing mode the user manipulates the document tree under the control of the tool. Amaya handles both the syntax and the structure, but the user has to know the semantics of the markup languages. The editor follows the DTD of each language. It creates only valid elements and attributes that are allowed in a given context. This is the mode in which the editing commands presented in section 4.3 are the most useful. This mode can be used for any markup language, like the previous two modes.

The last mode helps the user to manipulate structured elements according to their semantics. The Formatted view is well suited to this kind of interaction, as well as all the specialized views (which are often merged with the Formatted view). In this view, graphics is just manipulated as graphics, math as math, time as time (through a graphical metaphor). Full support is provided only for some languages, whose semantics are implemented in the editor, both in its formatting part and in its structure manipulation part. This allows Amaya to offer specific functions for editing efficiently XHTML tables, structured SVG graphics, complex mathematical expressions, SMIL animations, etc. according to their own semantics. This mode also enables sophisticated formatting of these elements in the main view, which gives users the feeling that they are manipulating the document directly, making the editor very easy to use, and relieving users from the details of the structure and the markup.

Semantics-driven editing requires specific code for implementing high-level editing operations. Therefore, it can be provided only for a few predefined languages, but some behaviours may be shared by several languages, or several parts of the same language. This is achieved in Amaya by associating some predefined editing behaviours with element names. Examples of shared behaviours are the commands used for editing XHTML tables and MathML matrices, whose structures are very close to each other, or the commands for setting hypertext links in several languages (XHTML, SVG, XLink) simply by pointing and clicking (see section 4.1).

4.5 Editing with style

The CSS style language plays an important role in editing complex documents. Style is an excellent way to let a user grasp the document structure. Although several XML languages allow some style properties to be included in the document itself through a `style` attribute, this is not considered good practice, and style information should be provided by separate style sheets, in external resources that are linked to the XML document. It is then easy to change the style of a document globally, just by associating different style sheets. Taking advantage of this mechanism, different style sheets can be specified to ease different tasks. In particular, style sheets can be designed specifically for the editing task. By showing very clearly all aspects of the document structure, they can help the user perform the editing task in the Formatted view.

It is important that an authoring tool be able to display the document under work with different style sheets, and the user should be able to develop new style sheets that fit his/her particular needs and preferences. Amaya allows the author to play freely with CSS style sheets. It offers a set of commands for disabling and enabling style sheets temporarily, to unlink a style sheet that is already associated with the document or to link a new style sheet. All these commands redisplay the document in the Formatted view according to the style sheets that are effective at that time. This is very efficient for choosing the right style sheet and for changing style sheet on the fly.

The fact that all objects included in a document, by transclusion or directly, are part of the same DOM tree, allows style sheets to apply to the document as a whole. Inheritance and cascading work on all parts of a document. If the font size, for instance, is changed on the document root, all included drawings and equations are changed in the same way as the rest of the document.

CSS is not only supported for editing and formatting compound documents. Amaya also provides functionality for creating, modifying and debugging style sheets. This functionality is fully integrated with the other editing features, thus allowing users to edit document content and structure at the same time as style, if they want to. This CSS editing functionality is available through a point-and-click interface for the most usual style properties, and through source code editing for all properties.

An interesting aspect of the CSS editing functionality in Amaya is debugging. The CSS parser reports errors with clickable messages that point to the relevant part in the erroneous style sheet, to help users to fix syntactic errors. But errors may also come from a complex combination of rules from several style sheets that finally set some unexpected style values. In addition, due to cascading and inheritance, style sheets for several markup languages may interact. In that case the difficulty is to locate the rule that was really selected by the cascading and inheritance processes. For that purpose, the user may select the element that looks strange and ask Amaya about the CSS rule that has set a given property. Amaya then shows what rule, in what style sheet, has set the value. This has proven very helpful for debugging complex sets of style sheets with sophisticated selectors.

5. CONCLUSIONS

In this paper we have reviewed some advances of XML as well as their impact on the techniques for editing structured documents. To take advantage of the new features brought by XML, we have made some proposals for enhancing XML editors and we have experimented these proposals, in particular in the Amaya Web editor.

The first set of proposals consists in making an XML authoring tool Web-aware. This improves significantly the us-

age of the Web related features of XML, and allows XML documents to be better shared and reused over the Web.

Another set of proposals is related to interaction modes and editing commands. We propose to go beyond the traditional context-sensitive menus or palettes for creating new elements in XML documents. We introduce faster and less constrained commands based on simple algorithms that rely on the DTD and on very little additional information. We also propose several modes of interaction that take into account the various abstraction levels of XML. Finally we extend structure editing by adding some semantics from the markup languages to provide the user with powerful and meaningful commands.

The convenience of the new editing commands is enhanced by the various views that are offered to users. With specialized views in addition to the usual views, users better comprehend the structure of a document and interact more easily with it.

The last proposals are in the area of integration. The idea is to handle compound documents by integrating the authoring features needed to process the different parts. This is achieved by maintaining a single internal representation of a compound document in a unique DOM tree. We also propose to integrate a style editing functionality with structure manipulation to give users the power of handling one more facet of structured documents in a consistent environment.

All editing techniques presented in this paper have been used and validated by the many users of Amaya. Being developed as an open source project since 1997, Amaya has benefited from an extensive feedback from its users. The features reported here have all been implemented during the last few years and then evaluated by the users. Many of these features, if not all, have been refined over the time thanks to the comments of the user community.

6. ACKNOWLEDGEMENTS

We are grateful to W3C for their support and contribution to the development and distribution of Amaya. We also acknowledge the valuable participation of a number of people in the work presented in this paper. In particular, we thank Laurent Carcone, Paul Cheyrou-Lagrèze, Pierre Genevès, Ramzi Guétari, Stéphane Gully, Jose Kahan, Daniel Veillard, and Daniel Weck. The whole Amaya community is acknowledged for their valuable contribution to the evaluation of the software.

7. REFERENCES

[1] *Amaya at W3C*. http://www.w3.org/Amaya/.

[2] S. Bonhomme and C. Roisin. Interactively restructuring html documents. *Computer Networks and ISDN Systems*, 28(7-11):1075–1084, 1996.

[3] B. Bos. Cascading style sheets home page. Technical report, http://www.w3.org/Style/CSS/.

[4] T. Bray, D. Hollander, A. Layman, and R. Tobin. Namespaces in xml 1.1. Technical report, W3C Recommendation, http://www.w3.org/TR/xml-names11/, 4 February 2004.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1.0 (third edition). Technical report, W3C Recommendation, http://www.w3.org/TR/REC-xml/, 4 February 2004.

[6] D. D. Cowan, E. W. Mackie, G. M. Pianosi, and G. de V. Smit. Rita–an editor and user interface for manipulating structured documents. *Electronic Publishing*, 4(3):125–150, April 1991.

[7] R. Furuta, V. Quint, and J. André. Interactively editing structured documents. *Electronic Publishing*, 1(1):19–44, April 1988.

[8] I. Jacobs. Architecture of the world wide web, first edition. Technical report, W3C Working Draft, http://www.w3.org/TR/webarch/.

[9] *LimSee2*. http://wam.inrialpes.fr/software/limsee2/.

[10] V. Quint. Systems for the manipulation of structured documents. In J. André, R. Furuta, and V. Quint, editors, *Structured Documents*, pages 39–74. Cambridge University Press, 1989.

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

[12] 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.

[13] *Xeena at Alphaworks*. http://www.alphaworks.ibm.com/tech/xeena/.

[14] *XML Spy*. http://www.xmlspy.com/manual/.