An XForms Based Solution for Adaptable Documents Editing

Istvan Beszteri
Telecommunications Software and Multimedia Laboratory, Department of Computer Science and Engineering, Helsinki University of Technology
P.O. Box 5400, FI-02015 HUT, Finland
Tel.: +358 50 482 1572
istvan.beszteri@hut.fi

Petri Vuorimaa
Telecommunications Software and Multimedia Laboratory, Department of Computer Science and Engineering, Helsinki University of Technology
P.O. Box 5400, FI-02015 HUT, Finland
Tel.: +358-(0)9-451 4794
petri.vuorimaa@hut.fi

ABSTRACT
The diversity of Internet enabled terminals is increasing continuously. Layout adaptation is becoming a crucial issue for the content providers. They have to support different client devices with minimal effort. This paper presents a document editing system that supports dynamic adaptation. It is implemented as a client-server application and uses cutting edge XML technologies.

Categories and Subject Descriptors
H.5.4 [Information Systems]: Information Interfaces and Presentation – Hypertext/Hypermedia.

General Terms

Keywords
Extensible Markup Language (XML), XFrames, XForms, XML database, XQuery, XUpdate, X-Smiles.

1. INTRODUCTION
Layout adaptation became a critical issue for the content providers in the last couple of years. At the beginning of the current century, mobile phones, handheld devices and even the home entertainment systems became capable to browse the Internet. This changed the former practice of document authoring to support a certain screen resolution.

As a straightforward reaction, content providers started to support the new devices by re-authoring the content for each supported device or device group. Even specific document description languages and protocols developed for the new devices. A good example was the appearance of the Wireless Markup Language (WML) [20] and the Wireless Application Protocol (WAP) [21]. WML supports WAP enabled mobile phones and handhelds by restricted set of graphical components, small document sizes, and as a consequence, low required bandwidth. It worked fine for a while, since there were not many devices to support at the beginning.

Later, when more and more Internet capable device appeared on the market, it became clear that re-authoring is not the right solution anymore. It requires too much effort to redesign a document for every supported client hardware and software. As a result of extensive research many layout adaptation methods were published. Bickmore et al [5] classifies the adaptation methods into five categories:

1. device-specific authoring
2. multiple-device authoring
3. client-side navigation
4. automatic re-authoring
5. web page filtering

It is easy to see, that automatic re-authoring requires the least effort and would be the most advantageous for content providers. Almost all the former research papers concentrated on solutions to adapt existing Hypertext Markup Language (HTML) [18] documents to different devices. These methods use the language elements of HTML to find the structure of the document. They are using a large set of heuristic rules for the adaptation. There exist a couple of impressive solutions, such as the WAP Gateway of Google [10], or Digestor [6].

The problems with this kind of methods are that they work only with a certain document description language, and since they are based on heuristic rules, they cannot be prepared for every situation. To solve this problem, the documents of the future, should support adaptation by providing extra information about themselves. This extra information should be well separated from the document content. This way the adaptation could be independent from the document description language. It provides a great flexibility for the actual document delivery, since the adaptation can be separated from the presentation.
eXtensible Markup Languages (XML) [7] are well suited to solve the problem above. By providing metadata, the necessary adaptation information can be added to the documents. Paper [2] presents a dynamic solution for layout adaptation. In this method, the document is decomposed to small document fragments, and glued to a whole, by using XFrames [14]. XFrames provides the necessary structural and sizing information about the document fragments. The whole adaptation works with the information provided by the XFrames tags, and does not deal with the actual document content. It makes the solution language-independent. However, it is the author's responsibility to fragment the document well enough, and provide the proper adaptation information.

To design the structure of such a document, and to provide all sizing data for the layout adaptation is a quite complicated task. Writing XFrames documents in a text editor is very error prone and boring. Paper [3] solves these problems, with an XForms based document editor. The idea is that the user edits the document by filling forms in a browser. When he or she submits the form, the browser generates the actual XFrames document from the form data. The problem with this solution is that it requires some workaround.

To edit XML documents with XForms, one has to provide the model of the output document. The model description rules, does not allow recursion. As a straightforward consequence, the model description of the output document. Although, it is possible to describe the following structure:

```
<xfm:instance>
  <a>
    <b>
    </b>
  </a>
</xfm:instance>
```

It means, that "a" can contain "b"s. The contained "b"s under "a" can contain other "b"s. But it does not mean that the "b"s under a "b" can contain "b"s again. The depth of the document is fixed to three. There is no rule to say that any "b" can contain other "b"s recursively.

Although, paper [3] gives a workaround for the problem, we were looking for a more sophisticated solution. The next sections present an XForms based document editor, which solves the problem above and provides convenient tools for generating adaptable documents.

2. CLIENT-SERVER ARCHITECTURE

In the following section we present a solution for editing adaptable web documents by filling XForms in a web browser. As it was discussed in the introduction, the main problem with using XForms to generate XML files is the lack of recursion in the model description of the output document. Although, it is possible to solve this problem on the client side, the solution has some drawbacks:

- It requires extra computing effort to run the browser.
- The client device has to be capable to run script interpreters or extra applications, which are not parts of the browser.

We can expect that most of the handheld devices will not meet these requirements. Either they lack computing power or they not necessarily are able to run the scripts, which extend the capability of XForms. The client side of the solution should be as simple as possible. It is much better if all the extra computation happens somewhere else than in the client device.

In this solution, we use a client-server application to eliminate the recursion problem. The main idea is the following. Instead of storing the architectural information of the edited document in the model description of an XForms document, an XML database is used on a server to store and update this information dynamically. The user communicates with the server by using XForms documents, but these documents are restricted to provide data for updating the document in the database. When the editing is ready the server generates an XFrames document from the corresponding data in the database.

The following lines represent an example of the output XFrames document:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<frames xmlns="http://www.w3.org/2002/06/xframes">
  <style type="text/css">
    group.page{width:100%; height:100%;}
group.mainlist{width:100%; height:100%;}
frame.banner{width:100%; height:11%; minwidth:99px; minheight:23px;}
  </style>
  <group id="page" class="page" compose="tabbed">
    <group id="mainlist" class="mainlist" compose="vertical">
      <frame id="banner" class="banner" source="http://localhost:8080/exist/frames/frameoptim/banner.html"/>
      <group id="menucontent" class="menucontent" compose="horizontal">
        <group id="menu" class="menu" compose="vertical">
          <frame id="menuxsmiles" class="menuxsmiles" source="http://localhost:8080/exist/frames/frameoptim/men u_x-smiles.html"/>
        </group>
      </group>
    </group>
  </group>
</frames>
```
is a basic object. It describes the actual content, and cannot be decomposed further.

At the very beginning the editing document contains a very simple model of the output document. It is a single group, which represents the top-level group of the output document that is a page. The application provides possibilities to add groups or frames to the architecture or remove them via XForms. Whenever such an editing happens, the server updates the XML file that stores the document data in the database, and regenerates the editing forms. This way we get a dynamic document model and there is no need for the recursion anymore.

Note, that this document model is stored on the server, instead of in the model description of the editing XForms document. However, it would also be possible to regenerate the same XForms document on the server side, by updating its document model dynamically. It would also be more sophisticated than editing the document on the client side, but using databases on the server side simplifies the implementation, and adds flexibility to the solution.

3. ADEDITOR

ADEditor (Adaptable Document Editor) is an implementation of the methods described in the former sections. It is implemented as a Java servlet. It uses an XML database to maintain the edited documents. Figure 1 depicts the architecture of the system.

Figure 1. Architecture of the document editing system.

ADEditor is integrated to Exist [13] via Cocoon [1]. Exist is an open source XML Database. ADEditor stores user data and the edited documents in the database provided by Exist. Exist also contains Cocoon, which is an XML based, open source web development framework, developed by the Apache group. Both Exist and Cocoon are implemented as servlets. In this implementation we used Tomcat servlet container to run them.

The client has to run an XForms compatible browser, in order to fill the forms sent by ADEditor. In the tests, we used X-Smiles [16]. X-Smiles is an open source XML browser, developed at Telecommunications Software and Multimedia Laboratory of the Helsinki University of Technology.

Figure 2. shows an example scenario of document editing.

Figure 2. Example of document editing with ADEditor.

In this example, first, the user accesses the login page of ADEditor, by sending a HTTP request to the server. Tomcat redirects this request to Exist, which forwards it to the ADEditor servlet. As an answer, ADEditor sends back a login form. ADEditor has readymade forms for login, group editing, frame editing, etc. These are static forms. ADEditor also generates dynamic forms from user data stored in the XML database of exist. After successful authentication, it generates the list of documents, which are created by the user and available for editing.

When a user registers to ADEditor, it creates a new collection into the XML database and stores the user authentication data in an XML file. The collection is identified by the username. When the user creates a new document, a new XML file is created in the user’s collection to store all the information about the document. The document description file is identified by a unique name, given by the user. After the successful authentication, ADEditor browses the collection of the user to generate the form of available documents.

After the user chooses a document for editing, ADEditor generates the form of the document tree by using the data in corresponding XML file. The data is acquired via the XQuery [17] API provided by Exist. The document tree is depicted with the help of a navigation list [18] in the form. From the document tree, the user can choose either a group or a frame for editing, by selecting a list item.
In the example scenario, the user selects a group for editing. The servlet generates a group editing form from the corresponding data in document description XML file. The data is acquired with XQuery like in the case of the document tree. With the group editing form the user can update the sizing information of the available space for the group. With the same form it is also possible to add new child groups and frames to the group. In this case, the user adds a new group.

When the form is submitted, ADEditor updates the document description XML file in the database via the XUpdate [12] API provided by exist. After the update, it regenerates the document tree form from the XML file and sends it back to the user. Finally, the user selects a frame for editing by selecting its list item in the navigation list of the document tree form. The servlet generates a frame editing form, similarly to the group editing form before. The user fills the form, and submits it to ADEditor. It updates the document description XML file, using XUpdate and sends back the new document tree. The editing can continue by selecting groups and frames again.

Since, all the necessary information about the edited document is stored in a single XML file in the DB, a simple XSLT [19] transformation can be used to generate the final XFrames document when the user is ready with the editing. The generated XFrames document can also be stored in the same collection as a snapshot.

To keep track of the editing phases, ADEditor is implemented as a state machine. Figure 3. shows a simplified version of the state machine.

For the sake of simplicity, Figure 3. does not contain the error handling states, and some less important transitions, such as the ones, pointing back to the Login or New User states from various other states.

The state transitions are triggered by XForms data, or events, submitted via HTTP POST or GET by the client. It can be seen that the actual document editing actions always start from and return to the Edit Existing Document state. When ADEditor enters this state, it always sends the updated document tree for the user as a navigation list. That is, it regenerates dynamically the document architecture according to the updated document model in the XML database, and updates the view of the model on the client side.

4. TEST AND RESULTS

As a test we re-authored the homepage of the X-Smiles project website. We made an adaptable document with the help of ADEditor. The document appears on desktop PCs similarly to the original page and it is also adaptable to any client hardware without modification. For the adaptation we used the same method that was published by I. Beszteri et al. in paper [4].

On the client side, we used the X-Smiles XML browser, on a conventional PC. Beside the PC layout, X-Smiles also can emulate various mobile devices and digital television. Figure 4. presents the original XHTML page rendered in the desktop PC view of the X-Smiles browser.
After creating a new user, and logging into ADEditor, we created a new document. The architectural view of the empty document is depicted on Figure 5. It only contains a top-level group that represents the whole page.

To achieve this layout, first we selected the top-level group, and added a frame to it. Then we selected it again and added the row. Finally, we selected it again and set the sizing information. Figure 7. shows the architectural view of the document at this stage.

We decomposed the original page the following way. The main page is treated as a column. It contains a frame that represents the banner, and a row. The row contains two columns. The first column holds the menu items on the left. The second column contains a row and two frames. The two frames are containing the release information text and the copyright text. The row is divided further to a column and a frame. The frame contains the X-Smiles picture. The column is separated to two frames, holding the version texts. Figure 6. shows the layout of the final document in the desktop view of X-Smiles browser.

We repeated the same steps recursively until we got the final architecture of Figure 6. Figure 8. represents the architectural view of the final document. Note that only the menu group is expanded. The frames are hid in all other groups. The frames of a group appear, when the user points the mouse pointer to the group.

Figure 5. Initial architectural view of new document in ADEditor.

Figure 7. Architectural view of document after adding a frame and a group.

Figure 8. Tree view of the final document.
After editing we generated the XFrames document by requesting ADEditor to save the document to that format. Then we tested the generated document by requesting it from different prototype views of X-Smiles. Figure 9. shows the results on three different clients.

Since the available drawing space was not sufficient, in all three cases, the adaptation algorithm had to split the original page to several pages. These new pages are represented by tabs in the browser. Figure 9. shows the same tab in all cases. In the case of PDA and digital television both the version column and the image fit to one page. In the case of mobile phone, the image got its own separated page. Note, that the image size also differs on the PDA from the image size on the phone.

5. DISCUSSION
It is the responsibility of the browser, how it represents the adapted pages. As it can be seen on Figure 9., in the current implementation, X-Smiles considers all the top-level objects in an XFrames document as pages, and represents them as tabs. It is not a problem if the screen is large enough like in the case of Figure 9. In small screens, if the document contains too many pages, the tab labels occupy a vast amount of the screen space.

It would be beneficial if the browser could guarantee the space for the pages. One solution could be to collect the labels to a separate navigation page. The navigation page could be represented by a menu, or by a navigation list. The menu items could hold internal links to the pages.

6. RELATED WORK
D. C. Brown et al. [8] published a paper about the effects of colour and layout adaptation on users’ task completion time. They concluded that both colour and layout adaptation greatly improves performance. Brusilovsky et al. [9] introduced AHA!, an adaptive hypermedia architecture. The layout description model of AHA! is similar to XFrames. It also builds up the document from compound objects and frames. H. Wu et al [22] presented a rule based adaptation model. This method is a Dexter-based [11] reference model.

7. CONCLUSIONS
In the previous sections we presented ADEditor. It is capable of editing adaptable documents. The edited documents can be represented in any XML document descriptor language, and they are embedded to XFrames. XFrames provides the architectural and sizing information for the adaptation. Since the adaptation works with the XFrames information only, the solution does not depend on the document description language itself. This also provides great flexibility for the adaptation, since it can happen at any point in the delivery chain.

ADEditor is implemented as a client server application. It uses XForms as an interface between the client and the server. It does not use XForms to generate the output document directly. Rather it collects the necessary data via XForms and stores them in an XML db. When all information is available, the output document can be generated on the server side with the help of XSL transformation. This way the problem of fixed model depth in XForms can be avoided.

Since all transformations and calculations done by the server, the client hardware has only an interface role. So the requirements to the client side are not high. The only criterion that the client has to match is the capability of XForms handling.
8. REFERENCES


