XML Active Transformation (eXAcT): Transforming Documents within Interactive Systems

Olivier Beaudoux
CER-ESEO
Angers, France
olivier.beaudoux@eseo.fr

ABSTRACT

Stylesheets and batch transformations are the most widely used techniques to transform "abstract" documents into target presentation documents. Despite the recent introduction of incremental transformations, several important features required by interactive systems are yet to be addressed, such as multiple sources (e.g. preferences and resources), multiple targets (e.g. multiple views), source-to-target linking (e.g. interacting with the source *via* the target), and bidirectional linking (e.g. interacting directly with the target). This paper proposes the use of *XML Active Transformations* (eXAcT) in order to fulfil these requirements. The eXAcT specification is based on the definition of two new DOM node types, *active fragment* and *anchor*, and on a transformation process inspired from XSLT. Our *jaXAT* implementation toolkit allows the active transformation of any DOM document into (but not limited to) SVG presentations.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; I.7 [**Document and Text Processing**]: Document Preparation—*Markup languages, Standards*

General Terms

Design, Languages

Keywords

Active transformations, XML, authoring tools, SVG, GUI

1. INTRODUCTION

XML technologies, such as CSS, XSL, SVG and namespaces, provide a wealth of capabilities for structured documents and authoring tools. However, traditional methods for editing compound documents are not sufficient to reap the benefits from these technologies [4]. An important challenge is to extend the process of transforming documents into presentation documents within interactive systems, as exemplified by recent incremental extensions to XSLT [3, 5]. However, other features required by interactive systems, such as multi-source and multi-target transformation, source-to-target linking, and bidirectional linking, are yet to be addressed. We propose the use of *XML active transformation* (eXAcT) in order to fulfil these requirements. eXAcT is defined as a DOM extension that specifies how source DOM documents can be transformed into target DOM documents. The *jaXAT* toolkit was developed in order to implement and test the eXAcT transformation process: the sources can be any DOM documents, the targets can be (e.g. SVG documents) and the transformation rules are defined as Java classes. The jaXAT toolkit[1] was initially developed in order to implement the document-presentation link of a more general model [1] that can be seen as an application of the MVC design pattern [2] to XML documents.

This paper presents eXAcT through a concrete example by addressing each feature of eXAcT (section 2), explaining the proposed *active fragment* and *anchor* node types (section 3), and illustrating rule programming (section 4).

2. XML ACTIVE TRANSFORMATIONS

An *XML active transformation* defines how source DOM documents can be transformed into target DOM documents and has the following features: *incremental transformation*, *multi-sources*, *multi-targets*, *target-to-source linking*, and *bidirectionality*. The eXAcT approach differs from incremental versions of XSLT [3, 5] in three ways. Firstly, the transformation process is based on observing the source and updating the target when needed by invoking *rule* methods. In the case of a bidirectional transformation, the process also observes the target document and updates the source by invoking *inverse rule* methods. Secondly, eXAcT specification is a DOM extension (it uses the IDL specification language) which can be implemented by using various object programming languages. Finally, fragments instantiated by transformations are created by using external tools adapted to their format (e.g. a drawing tool for SVG fragments).

Figure 1 illustrates a typical use of eXAcT. The eXAcT process transforms DOM source documents (1) into a DOM target document (e.g. a SVG document) (3). Multiple eXAcT transformations can provide multiple views of a common source. A transformation also refers to resource documents for the definition of fragments and symbols it instantiates (2). The source differs from the resource by its incremental capability: eXAcT updates the target *only* when the source changes. Views (4) display targets and allow user interaction. Since eXAcT operates on DOM nodes, reverse links from target nodes to source nodes can be easily provided (5). This allows the selection of source nodes from the target views, so that users can

[1]http://www.eseo.fr/~obeaudoux/jaxat

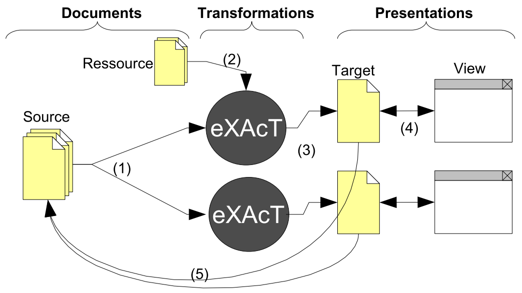

Figure 1: XML Active Transformations

interact with the source document. In addition, this allows the implementation of generic unidirectional and bidirectional rules that can be used in many contexts, thus reducing the rule coding effort.

We developed the jaXAT toolkit that provides a full implementation of the eXAcT specification in Java. The implementation is based on Apache libraries[2]: Xerces + Xalan (DOM), and Batik (SVG). It also provides a faster viewing through our Simple SceneGraph (SSG) implementation. We are currently working on an HTML and an X3D viewer.

3. DESIGNING ACTIVE FRAGMENTS

In order to explain eXAcT, we will use a simple example. The main document source, called dirTree, represents a local directory tree as follows:

```
<dir> <name>root dir</name>
     <dir  id='dir1'> <name>dir 1</name>
        <dir  id='dir2'> <name>dir 2</name>
           <file> <name>file A</name> </file>
           <file> <name>file B</name> </file>
        </dir>
        <file> <name>file C</name> </file>
        <file> <name>file D</name> </file>
      </dir>
      <dir  id='dir3'> <name>dir 3</ name> </dir>
      <file> <name>file E</name> </file>
</dir>
```

The eXAcT process consists of transforming this tree into an SVG view (see figure 2). Users can expand or collapse <dir> elements of the dirTree document through this view. A second source document, called dirState, contains the expanded/collapsed information of <dir> elements within the state attributes, as follows:

```
<dir  ref='dir1' state='expanded'/ >
<dir  ref='dir2' state='collapsed'/ >
<dir  ref='dir3' state='collapsed'/ >
```

The transformation is based on the combination of *active fragments* and *anchors* (figure 2). The directory-frame active fragment is first inserted as the root fragment of the target. It defines both the graphical representation of the frame and two anchors. The name anchor defines the location of the text that will display the name of the viewed directory (here "root dir"), while the contents anchor defines the location of the directory contents. For each anchor, eXAcT defines a *context source node*, an XPath *expression*, a *rule*, and an optional *reverse rule*. For example, the name anchor of the directory-frame fragment is managed by a GenericRule instance, with a dir/name/text() XPath applied to the root of the dirTree document. When the anchor is initially created, or each time the context

[2]http://xml.apache.org

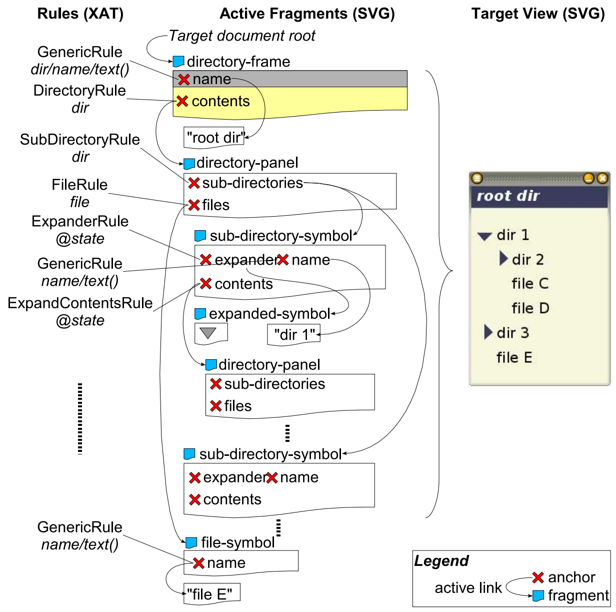

Figure 2: Active fragment design

source node changes, the eXAcT process applies its XPath *expression* to the *context source node*. The *rule* instance is then called on *each* node that matches the XPath. In the case of the name anchor, the GenericRule creates a child active fragment (see section 4) containing a text node that holds the name of the root directory (*"root dir"*). The DirectoryRule instance, associated with he contents anchor, creates a directory-panel active fragment that defines two anchors holding a representation of the sub-directories and the files. The sub-directories anchor manages the insertion of two children sub-directory-symbols representing "dir 1" and "dir 2". Respectively, the files anchor manages one child file-symbol representing "file E". The transformation process then continues recursively. The second dirState document comes into play at the ExpanderRule and Expand-ContentsRule level: the state XPath expression is applied within the dirState document (see section 4). The combination of active fragments and anchors allows the definition of active transformations through a recursive process in a way inspired from XSLT. However, the main difference is that generated fragments are located within the DOM target tree, thus allowing their management by the associated anchors. The third document, called svgRes, holds the definition of these active fragments. They have been drawn using Inkscape[3] as <g> group elements. For example, the directory-frame fragment is defined as follows:

```
<g  id='directory-frame'>
   <g>   <!– title bar –>
      <rect fill="gray" .../>
      <text ...>
         <xat:anchor   name='name'/>Dir name<xat:anchor/>
      </text>
   </g>
   <g  ...>   <!– contents –>
      <xat:anchor   name='contents'/><xat:anchor/>
   </g>
</g>
```

The title bar is a group element (<g>) containing a gray rectangle followed by a text. The text holds the name anchor (accessible through the xat prefix), and the next group element defines the location of the directory contents.

[3]http://www.inkscape.org

4. PROGRAMMING RULES

The transformation is initiated by the following Java code:

```
dirTree  = getDOMImpl().loadDocument(”dirTree.xml”);
dirState  = getDOMImpl().loadDocument(”dirState.xml”);
svgRes  = getSVGImpl().loadDocument(”svgRes.svg”);
target  = getSVGImpl().createDocument();
root  = target.createActiveFragment(dirTree,
        svgRes,  ”directory-frame”);
root.getAnchor(”name”).activate(dirTree, ”dir/name/text()”
        new GenericRule(), new GenericInverseRule());
root.getAnchor(”contents”).activate(dirTree, ”dir”,
        new DirectoryRule());
target.appendChild(root);
canvas  = getSVGImpl().createCanvasView(target);
canvas.addInteractor(new  ExpandInteractor());
```

The dirTree, dirState, and svgRes documents are loaded, and the target is then created as an empty SVG document. Its root active fragment is created from the directory-frame symbol of svgRes. Its name and contents anchors are then retrieved and *activated*. The activation process consists in calling the anchor rule whenever needed, as explained in the previous section. The active fragment is appended to the target, and a canvas view is then created in order to display the target document. A user can interact with the view by using the ExpanderInteractor instance that expands or collapses any clicked directory. Each rule (and inverse rule) is defined by implementing the eXAcT Rule (respectively InverseRule) interface. For example, the contents anchor of the root active fragment is handled by the elementInserted method of the DirectoryRule class, as follows:

```
public void  elementInserted(Anchor  anchor, Element  dir) {
     ActiveFragment  f  = target.createActiveFragment(
          dir, svgRes,  ”directory”);
     f.getAnchor(”sub-directories”).activate(dir, ”dir”,
          new SubDirectoryRule());
     f.getAnchor(”files”).activate(dir, ”file”, new FileRule());
     anchor.appendChild(f);
}
```

The anchor argument represents the “calling” anchor that requires updating its child fragments as soon as the dir element is inserted into the dirTree source document. The active fragment f is created from the directory SVG symbol. It activates both anchors named sub-directories and files (see figure 2), then is appended to the calling anchor. Each rule coding follows a similar scheme: creating active fragments, activating their anchors, then appending fragments to the calling anchor. Moreover, the *same* xxxInserted method is used for both the initial construction of the target and its subsequent updating. Rather than implementing the Rule interface, eXAcT transformations usually extend or directly use the GenericRule class that provides useful generic features. For example, the *name* anchor of the directory-frame fragment is managed by the generic rule. It performs a “copy text” operation from the source text node to the target text node, so they always remain equal. Moreover, GenericRule provides an implementation of the “remove node” operation by implementing the xxxRemoved methods. It consists in removing an active fragment whenever the source node that created it has been removed. The following code excerpt illustrates the GenericRule implementation:

```
// generic “copy text”
public void  textInserted(Anchor anchor, Text t) {
     Document d = anchor.getOwnerDocument();
     Node f = d.createActiveFragment(t, t.getNodeValue());
     anchor.appendChild(f);
}
public void  textChanged(Anchor anchor, Text t) {
     Node f = anchor.getChildFragments().item(0);
     f.getChildNodes().item(0).setNodeValue(t.getValue());
}
// generic element removal
public void  elementRemoved(Anchor anchor, Element e) {
     // search for the fragment linked to source e
     NodeList children = anchor.getChildFragments();
     for (int n = 0; n < children.getLength(); n++)  {
        ActiveFragment f = (ActiveFragment) children.item(n);
          // and remove it from the anchor
        if (f.getSourceNode() == element)
          anchor.removeChild(f);
     }
}
```

For example, SubDirectoryRule (see below) and FileRule inherit from GenericRule and refine it by only updating the SVG location of each node. Generic behaviors are also provided by the GenericInverseRule. It defines inverse “copy text” and “remove node” operations (not detailed here). The dirState document enters into action within the SubDirectoryRule rule that creates a new branch, with dirState document as context nodes, as follows:

```
public void elementInserted(Anchor a, Element  dir) {
     String id = dir.getAttribute("id");
     Node  refDir  =  dirState.getNodeByPath("// dir[@ref='" + id + "']");
     ActiveFragment f = targetDoc.createActiveFragment(dir,
         svgRes, "sub-directory-symbol");
     f.getAnchor("sub-directory- expander").activate(refDir,
          "@state", new ExpanderRule());
     f.getAnchor("sub-directory- name").activate(dir,
          "name/ text()", new GenericRule());
     f.getAnchor("sub-directory- contents").activate(refDir,
          "@state", new ExpandContentsRule());
     a.appendChild(f);
     updateLocations();
}
```

5. CONCLUSION

This paper proposes the use of XML Active Transformation (*eXAcT*) as a transformation process adapted to interactive systems. The transformation model is defined as a DOM extension that combines *active fragment* and *anchor* nodes in a recursive process inspired from XSLT. eXAcT offers mechanisms for incremental transformation, multi-source and multi-target documents, target-to-source linking, and bidirectional linking. Moreover, the proposed *jaXAT* implementation toolkit allows the use of dedicated tools (e.g. a SVG drawing tool) for designing active fragments, and the use of an object programming language (Java) for coding rules and for debugging. The next steps consist in using a faster XPath processor (e.g. Jaxen), and in comparing the effectiveness and ease-of-use of eXAcT with incremental XSLT. eXAcT and the jaXAT toolkit are a subset of a general project [1] that aims at defining new foundations for Web document interaction.

6. REFERENCES

[1] O. Beaudoux and M. Beaudouin-Lafon. DPI: A conceptual model based on documents and interaction instruments. In *Proc. of IHM-HCI'01*, pages 247–263. Springer Verlag, 2001.

[2] G. E. Krasner and S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of OOP*, pages 26–49, 1988.

[3] M. Onizuka, F. Y. Chan, R. Michigami, and T. Honishi. Incremental maintenance for materialized XPath/XSLT views. In *Proc. of WWW'05*, pages 671–681. ACM Press, 2005.

[4] V. Quint and I. Vatton. Techniques for authoring complex XML documents. In *Proc. of DocEng'04*, pages 115–123. ACM Press, 2004.

[5] L. Villard and N. Layaïda. An incremental XSLT transformation processor for XML document manipulation. In *Proc. of WWW'02*, pages 474–485. ACM Press, 2002.