An Incremental XSLT Transformation Processor for XML Document Manipulation

Lionel Villard
Opéra Project
INRIA Rhône-Alpes Research unit
Zirst - 655 avenue de l'Europe - Montbonnot
38334 Saint Ismier Cedex
France.
Tel: +33 (0)4 76 61 53 82
Fax: +33 (0)4 76 61 52 07
lionel.villard@inrialpes.fr

Nabil Layaïda
Opéra Project
INRIA Rhône-Alpes Research unit
Zirst 655 - avenue de l'Europe - Montbonnot
38334 Saint Ismier Cedex
France.
Tel: +33 (0)4 76 61 53 84
Fax: +33 (0)4 76 61 52 07
nabil.layaida@inrialpes.fr

ABSTRACT

In this paper, we present an incremental transformation framework called *incXSLT*. This framework has been experimented for the XSLT language defined at the World Wide Web Consortium. For the currently available tools, designing the XML content and the transformation sheets is an inefficient, a tedious and an error prone experience. Incremental transformation processors such as *incXSLT* represent a better alternative to help in the design of both the content and the transformation sheets. We believe that such frameworks are a first step toward fully interactive transformation-based authoring environments.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors – *Interpreters, Optimization.*
D.2.6 [**Software engineering**]: Programming Environments - *Interactive environments.*

General Terms

Algorithms, Performance, Design, Experimentation, Languages, Theory.

Keywords

XML, XSLT, Incremental transformations, Authoring tools.

1. INTRODUCTION

The advent of the XML standard at the World Wide Web Consortium has triggered the definition of an incredible amount of vocabularies in different areas of content representations. Although these vocabularies are defined separately and designed for a variety of purposes such as content structure descriptions, layout languages, vector graphics or mathematical formulas rendering, they can also be inter-mixed in a single and same document. These vocabularies share the same encoding language at the syntactic level thanks to XML and are combined using another XML companion standard called XML Namespaces [12]. In most cases however, the content is encoded using third party DTD's while the resulting documents are encoded using rendering vocabularies such as XSL [7]. The result is an increasing diversity of document classes and vocabularies and a lack of authoring tools that copes with this diversity.

In this paper, we propose an incremental transformation framework, called *incXSLT*, which can be used, in particular, in the editing of XML documents through one or many of its rendered presentations. These presentations are described as XML markup and produced usually through a transformation process. In order to facilitate this operation in an interactive authoring system, we propose to extend transformation processors to be the basis of XML documents manipulation. Authoring is one particular use of such a framework and other applications can be the tuning of XSLT transformations, the fast updated of large web sites or the design of XML Schemas. This paper focuses on how to achieve incremental updates to the presentation after a source XML document modification occurs.

The paper is organized as follows: in the following section, we motivate the need of incremental transformations and we discuss about related works. In the fourth section, we describe the general architecture of transformation based authoring systems. Then, we identify the main characteristics of the XSLT transformation language and we compare it with other available languages. In the fifth section, we describe our incremental transformation processor *incXSLT* that is the central part of the proposed framework. This description is completed with an evaluation of the current implementation. In the last section, we give some conclusions and draw some perspectives related to editing transformation sheets.

2. GOALS

In the currently available authoring systems [2][19], the only way to edit XML documents is through a lower-level text representation or at most through an enhanced representation as a graphical tree. Some authoring tools [18] give the user the ability to attach style to XML elements. This association simplifies the authoring of a document by making the XML content more accessible to the user through the graphical interface. Even though, style sheets remain of a very limited help when considering more complex presentations. More recently, the **<xsl>Composer** [22] allows the authoring of XSLT transformations by direct manipulation. However, this tool has many lacks: in particular the transformation process is executed from scratch after each modification of the transformation sheet. The result is an increasing processing cost proportional to the size of the document.

WWW 2002, May 7-11, 2002, Honolulu, Hawaii, USA.
ACM 1-58113-449-5/02/0005.

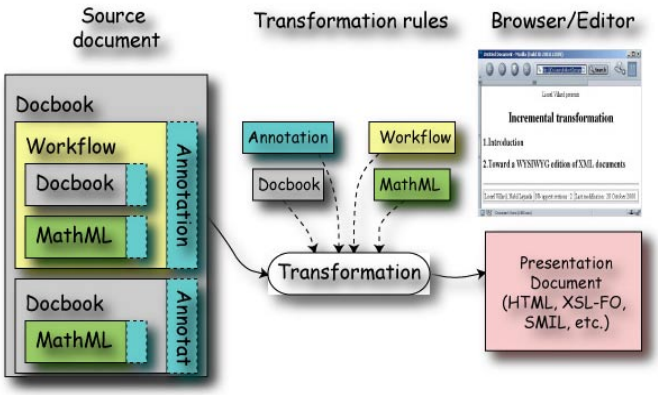

Figure 1. Presentation process

A presentation of an XML document is generally obtained using the production process described in Figure 1. An XML document, which can be composed of many vocabularies, is transformed to a vocabulary closer to its final presentation. The obtained document is then formatted and graphically rendered to the reader.

In general, one cannot make strong assumptions about the documents that are to be edited by the user, nor what the user is trying to achieve with the XML content. For example, the author may be willing to design a transformation sheet that allows the production of the layout for a particular class of documents. Due to the diversity of the source document classes, it's necessary to help the author creating transformation rules that handles the rendering of the different XML elements.

In the case of applications related to the content creation at the source XML level, the incremental transformation framework can help in achieving several functions. A first use can be the design of a function that provides a graphical preview of the document layout. These documents can be created from scratch or, in a more productive manner, updated incrementally after a modification in a source representation of the document occurs. Another application can be the deployment of XML content toward devices with different capabilities and users with different preferences. In this case, the transformation processor can be used to provide navigation through the various presentations corresponding to the different devices and users. The navigation can be achieved simply by selecting the target device and user profiles [5]. At the transformation level, this is equivalent to a change in the transformation rules and parameters sets.

As we have seen, the transformation framework can make the authoring much more efficient and reliable: the author can check the content design directly on the final presentations. But still, editing directly the source document remains an inefficient, a difficult and an error prone task. We believe that an additional step toward a much more comfortable edition of XML documents is to get closer to an interactive approach and to provide high level editing functions related to the document domain.

One of the most important key aspects for the success of such frameworks is related to performance. In order to be usable, the transformation process must be fast enough for the user operations. In particular, it is critical that modifications of the source document or the transformation sheets are reflected promptly to the user. In these situations, making transformation programs incremental becomes a major issue. Incremental changes allow controlling the scope of the document changes without requiring a global re-evaluation of the entire transformation. Controlled incremental updates are capable of efficiently updating the result of a computation when the source document or the transformation rules changes slightly. Therefore, incremental change handling becomes a valuable help for a user designing XML content or adjusting the transformation sheets.

In this paper, we focus on the target document updates occurring as a result of changes in the source document. This paper does not cover reverse transformations required when the changes are applied in the target document and are to be reflected in the source document. Examples of application that achieves such bi-directional changes can be found here [15][18]. Compared to our framework, most of these applications maintain these changes by restricting the source and the target documents to have an almost identical structure. Of course, this assumption is not relevant for XSLT-like transformations where there are no constraints on the transformations.

3. RELATED WORKS

In the field of programming languages, a significant amount of work has been achieved on incremental computations [16]. In most cases, they either handle specific problems for particular input changes in the programs, or at the other end propose too general frameworks. Even though, some specific techniques and frameworks have helped in the design choices we have made for *incXSLT*. For example, selective re-computation used also in [17] helped in the identification of the rule fragments responsible for the modifications between source and target elements. The second technique is based on performance optimization based on intermediate result caching [10]. In our case, we reused this notion of local caches to allow execution state restorations for transformation statements. However, one of the problems we encountered with caches is the determination of the caches size. In the particular area of incremental processing of XSLT transformations, there is no experience on this subject reported in our knowledge.

4. DESCRIPTION OF XSLT

4.1 Why XSLT?

In the literature, several transformation languages have been proposed for a variety of purposes. Balise [4] for example, is a script language in which some functions of tree manipulation (creation and copy) are provided. Omnimark [11] is another language more suitable for text streams programming. An Omnimark program consists of rules that define events such as general markup events produced when parsing an XML document. XSLT [11] is a functional language especially designed for XML document transformation. An XSLT program or transformation sheet consists of transformation rules (templates) associated with patterns. When a rule pattern matches the source document being processed, the corresponding rule is instantiated and creates as a result a tree fragment.

The transformation power of these three languages is, in fact, quite similar. XSLT has been designed as a side effects free language. A function in a programming language is said to have side effects if it makes changes to its environment, for instance, if it modifies a global variable that another function can read. Functions that have no side effects can be called any number of times and in any order. This property is crucial for the implementation of an efficient incremental transformation engine. As we will see later, restoring as fast as possible the execution state (processor context for XSLT) for a given statement (or instruction) is a key point for

efficient incremental transformations. Computing this state for a language with side effects is more expensive in time and space than a side effect free language [11][1].

In order to present our incremental transformation processor, the following section describes the concepts related to XSLT in the context of incremental processing.

4.2 Main concepts

As described earlier, XSLT is a language specifically designed for transforming the structure of an XML document. The transformation processor (see Figure 2) takes as input an XML document and transforms it by finding first a transformation rule matching the root node. In a second step, it executes sequentially the instructions contained in the rule. The result is called the target document.

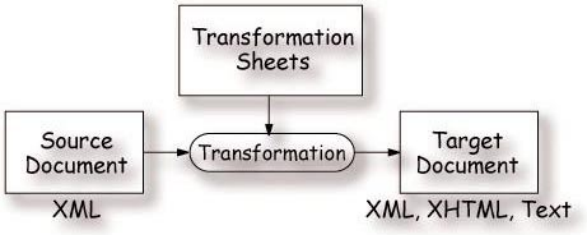

Figure 2. Transformation process

4.2.1 An example

Before getting into the details of the *incXSLT* processor, we give first a working example that we will use throughout the remaining part of this paper. Consider the following source fragment document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<article>
  <title>Incremental transformation</title>
  <artheader>
    <authorgroup>
      <author>
        <firstname>Lionel</firstname>
        <lastname>Villard</lastname>
      </author>
      <author>
        <firstname>Nabil</firstname>
        <lastname>Layaïda</lastname>
      </author>
    </authorgroup>
    <date>28 October 2000</date>
  </artheader>
  <section>
    <title>Introduction</title>
    <para>…</para>
  </section>
  <section>
    <title> Toward a WYSIWYG edition of XML Documents</title>
    <para>…</para>
    <section>
      <title>An example</title>
      <para>…</para>
    </section>
    <section>
      <title>Process overview</title>
    </section>
  </section>
</article>
```

This XML document is an instance of the docbook [13] document class. It represents an article composed of global data, gathered under the artheader element, such as the author names. The content of article is organized in sections. A possible presentation on the screen is illustrated in Figure 3. It is followed by the corresponding source code of the target representation as HTML. Figure 3 shows the table of content of the article below the first author's name and the article's title. At the bottom of the screen, the complete list of authors is rendered, followed by the number of higher-level sections and the document last modification date.

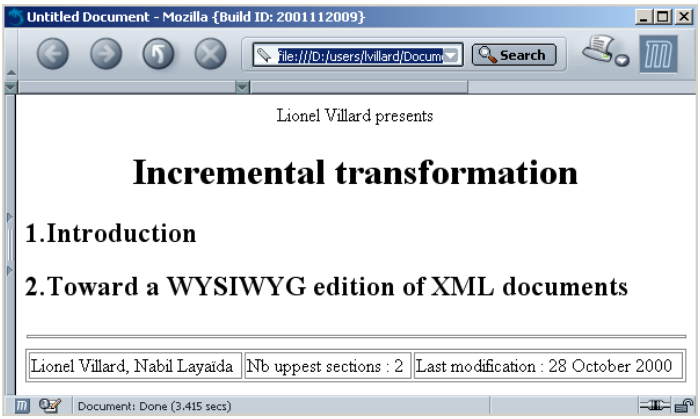

Figure 3. A presentation of a docbook document

```
<html>
  <body>
    <p align="center">Lionel Villard  presents</p>
    <h1 align="center">Incremental transformation</h1>
    <h2 align="left" style="padding-left=0px">1. Introduction</h2>
    <h2 align="left" style="padding-left=0px">2.Toward a WYSIWYG edition of XML Documents</h2>
    <hr>
    <table width="100%" border="1">
      <tr>
```

```
      <td>Lionel Villard and Nabil Laya&iuml;da</td>
      <td>Nb uppest sections : 2</td>
      <td>Last modification : 28 October 2000</td>
    </tr>
  </table>
 </body>
</html>
```

The presentation above is the result of the transformation applied by the transformation sheet given in annex A. A fragment of the execution tree of this transformation is given in Figure 4. At the beginning of the transformation, the following template rule is instantiated on the document root element:

```
8.  <xsl:template match="article">
9.  <html>
10.   <body>
…       <!-- header generation : "Lionel Villard
              presents" -->
17.     <h1 align="center"><xsl:value-of
        select="title"/></h1>
18.     <xsl:apply-templates select="section"> …
        </xsl:apply-templates>
…       …
22.     <table border="1" width="100%">
…           …
28.     </table>
29.   </body>
30.  </html>
31. </xsl:template>
```

In the beginning of the transformation, html and body elements are copied in the target, followed by the header. The title is presented inside an h1 tag. Then, the table of content is generated thanks to the instruction line 18. This instruction selects sections to be processed (see below). The html table containing the full list of author names, the number of higher-level sections and the date of the last modification are then generated.

When processing a section, a template that best matches the section node type is searched in the transformation sheet. In our case, the following template is instantiated with a parameter that will be used to indent the section entries. The current source node is the section node being instantiated. This node will be used to evaluate expressions. For example, the expression title line 53 retrieves the title of the section being instantiated.

```
33. <xsl:template match="section">
34.   <xsl:param name="indent">0</xsl:param>
35.
36.   <xsl:variable name="heading">
…       <!-- Depending on the section depth, choose a
              heading tag -->
43.   </xsl:variable>
44.
45.   <xsl:element name="{$heading}">
46.     <xsl:attribute name="style">padding-left=
47.       <xsl:value-of
            select="$indent"/>px</xsl:attribute>
…       …
51.     <xsl:number value="position()" format="1."/>
…       …
53.     <xsl:value-of select="title"/>
54.   </xsl:element>
55.
56.   <xsl:if test="count(ancestor::section) &lt;
                    $toc.depth - 1">
57.     <xsl:apply-templates select="section">
…       …
61.   </xsl:if>
62. </xsl:template>
```

A variable named heading is firstly created (line 36). Its value contains the name of the heading tag that depends on the section depth (h2 for level 0, h3 for level 1, etc.). Then an element with the name contained in the heading variable is generated (line 45). An attribute named style is added to this element. Its value defines a left padding of indent pixels. The content of the element previously generated corresponds to the position of the section relative to its parent (line 51). It is followed by the title content of the section (line 53). Then, depending on the section depth (line 56), children sections are processed.

Now that we have presented the principle of this transformation sheet, we illustrate the transformation through a simple authoring scenario. For example, think of an author that modifies the source document by inserting a new section element to article element in the previous example. In this case, the consequences on the target document are the following: an h2 element must be generated with the corresponding number, and depending on where the section is inserted, the following section numbering must be updated. The counter of higher-level sections must also be updated in the html table. In summary, the list of instructions that need to be re-executed is the following:

- The apply-templates instruction that select section elements (line 18): the whole template that matches section elements must be applied with the new inserted section as the source node.
- The number instruction for all sections following the newly inserted section element (line 51).
- The second cell of the html table (line 25).

In the remaining part of this paper, we will explain how to determine the list of previous instructions, and how to update the target document.

4.2.2 Overview of the incremental processing

One of the goals of an incremental processor is to change only target document fragments that need to be updated. At the transformation sheets level, this is equivalent to selecting instructions that need to be re-executed. Source nodes for which this re-execution applies must also be identified. The method to perform selective transformation relies mainly on XPath [23] expression structure analysis. This analysis must be carried out as a pre-processing stage of the incremental session. Then, for each of theses instructions, their execution state must be restored first in order to be able to execute them. As the state must be restored as fast as possible, caching techniques are used in *incXSLT*. As stated previously, one of the problems we have been facing is the determination of the minimum data that needs to be saved. The instruction is executed using the incremental version that we have developed instead of their non-incremental counterpart available in the XSLT batch processors.

4.2.3 Expressions and patterns

In XSLT, a number of instructions use expressions in attribute values. An expression is generally composed of one or many path expressions. A *Path expression* defines a navigation path through the navigation tree. When the path expression is evaluated, the result is a set of source nodes. For example, the expression artheader/authorgroup (line 24) is a path expression. The evaluation of this expression is a set of authorgroup elements with artheader element as a parent.

The evaluation of such expressions depends on a static context and a dynamic context [9]. For incremental transformations, only the dynamic context need to be restored each time an instruction is executed. This context depends on the state of the processor at the time the expression has been evaluated. This context consists of:

- The current values of all variables that are in the scope of the expression.
- The current node: this is the node in the source document that is currently being processed.
- The current node list: when an apply-templates or for-each instruction are used to process a list of nodes, that list becomes the current node list.
- The current position indicates the position of the current node in the current node list.

The expression syntax is also used to specify patterns. A pattern is a particular expression with some restrictions:

- The result type of the pattern evaluation must be a node set.
- Only child and attribute axes are permitted.

During an incremental session, the instructions that need to be re-evaluated are those whose associated expression(s) can potentially change. In particular, this occurs when an attribute is modified in the source document. In the general case, the result of an expression can change either because its evaluation context and/or the result of expression's location paths has changed. For example, the instruction <xsl:value-of select="position()"> needs to be re-evaluated when the position of the current node changes. In this case, the position of current node changes only when a section is added or removed before the current node.

As said earlier, a path expression selects a node-set. In most cases, the nodes types included in this node-set can be determined without the knowledge of the dynamic context. For example, the result of the expression article/section depends only on the article and section elements. By taking advantage of such a property, we use it to make a first filter to remove unnecessary instructions that does not need to be re-evaluated.

In fact, the nodes in the selected node set match a particular pattern. This pattern is obtained from the path expression. For this pattern, it is necessary to remove all dynamic context references. This operation can be quite complex. For example, the reference of the toc.depth top-level parameter in the expression line 56 (count(ancestor::section) < $toc.depth - 1) must be de-referenced in order to analyze the parameter value. As the toc.depth parameter does not contain any location paths and any dynamic context references, this expression matches only the pattern section. More complex cases can occur, especially when the expressions contain references to template's parameter.

Compare to the XSLT definition of patterns, some restrictions were lifted in order to remain closer to the definition of this node set during the conversion. All the possible axes were permitted except ancestor, ancestor-self, following and preceding. The restriction of axes in XSLT has been introduced for performance reasons and the goal was to allow efficient pattern matching. But in our case, pattern matching occurs relatively less frequently than during a batch transformation. Secondly, having a more accurate selection in an incremental transformation allows minimizing the instructions that need to be re-evaluated. However, as the selection must remain reasonably efficient, ancestor, ancestor-self, following and preceding axes are still not permitted because of their poor performances. For example, testing a node against the pattern slide/ancestor::title requires a navigation through all the descendants of the currently modified element (of type title) in order to retrieve an element of type slide.

Given that the dynamic context cannot be known beforehand, in particular local variable values, their use in patterns is forbidden. Later in this paper the conversion algorithm will be given in more details in section 5.1.

4.2.4 Instructions

During an incremental transformation session, the dynamic context must be restored in order to perform the newly introduced modification. The target context of the resulting document must be restored also. In order to understand how the different XSLT instructions affect the transformation process, they have been classified under a set of categories. This will help later in defining the data that needs to be stored in the cache. Global instructions (attribute-set, namespace-alias, etc.) are static parameters executed at the beginning of the transformation and not depend on the source document. Modularization instructions (import, include, etc.) define how transformation sheets are physically organized, so they do not depend on the source document. Variables (variable, param) instructions allow defining global or local variables and template parameters. The variables are a significant part of a processor context (see the previous section). Flow instructions (apply-templates, for-each, if, etc.), allow the control of the transformation execution and in particular the choice of source node to be instantiated (apply-templates instruction). Producer instructions (value-of, element, etc.) generate fragments of the target document. These instructions are the bridge between transformations and the target documents and are very valuable in restoring the target context.

4.2.5 Execution flow tree

During a transformation, the instructions are executed sequentially. These instructions will perform some actions and depending on the previous classification some data will need to be stored. The execution flow tree is a representation of the instructions execution. Many systems have used variants of the execution flow tree mainly in the context of side-effect languages [1][11]. The goal of theses systems is to provide execution backtracking facilities in order to help program debugging: during the debugging phase, the user can undo the execution of some instructions, change program inputs and re-execute the program incrementally. The incremental execution relies on the tracing of program execution to create history logs (list of couples (line, variables values)). From theses logs, the memory state can be restored and the re-execution can be achieved. The main problem in these systems is that the time and space costs can be prohibitive. In the transformation context, the problem varies according to the following aspects:

- The execution of the transformation is generally bounded. The size of history is much smaller.

- Not all the execution history needs to be stored. As the language is side effect free, restoring dynamic context such as variable values can be easily and rapidly achieved.
- The incremental transformation takes place after an entire initial transformation has been performed.
- The modification of transformation sheets (the program) during the incremental session is possible.

In our system, when the user starts a transformation session, the execution flow tree is built in a batch mode first. In order to update incrementally the source document or the transformation sheet, we need to determine exactly what to store in the tree. Storing all of the expression evaluations, the template instantiated for a source node and the links to the target document would require a huge amount of memory.

After an editing operation, some instructions need to be re-executed for a given source node. If we suppose that the processor is able to determine this set of instructions, executing them would require the minimal processor context (the part which affects their execution) as well as the target context. In order to set the processor context and the target context from any instruction we need to traverse up the execution flow.

Figure 4 represents a fragment of the transformation described earlier in section 4.2.1.

The execution flow tree structure is composed of so-called *execution nodes*. Execution nodes of type **flow** contain the value of their associated expression. For instance, apply-templates execution nodes have template nodes as direct children. Template nodes have links to the source nodes. Therefore, from apply-templates nodes we can retrieve source nodes that composed the context node list. **Producer** nodes contain data related to the target tree. This data will serve to restore the target context. For example, the instruction element has a link to the element it generates. For a character producer such as value-of instruction, only the number of generated characters needs to be stored. Execution nodes that really need to be stored depend on the incremental execution optimization used during the incremental session (see section 5.2).

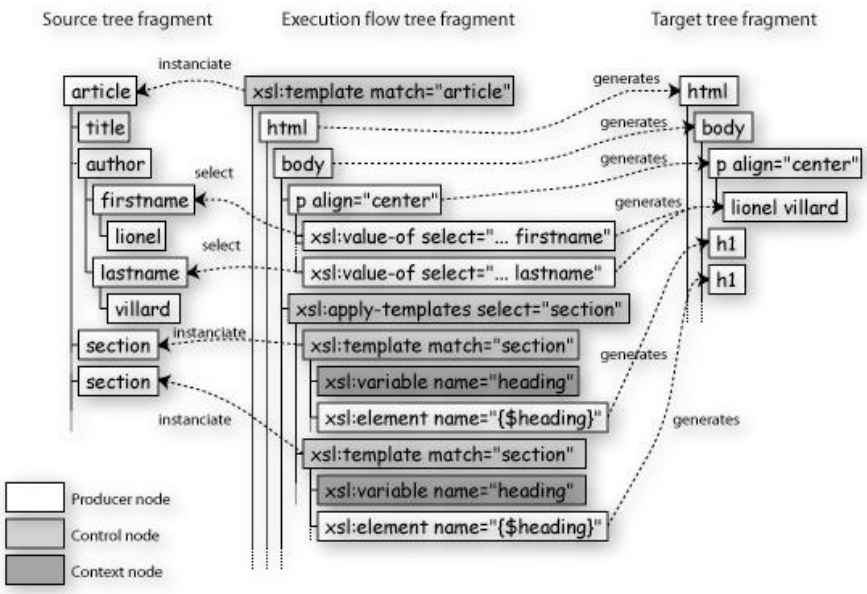

Figure 4. The execution flow tree of paper example.

The data structure presented here allows having at any instant of an incremental transformation session the source and target contexts. Even if we need some extra processing to obtain for example, variable values, this data structure is the key data representation of our *incXSLT* incremental processor.

5. INCREMENTAL TRANSFORMATION

In this section, we describe how to select the instructions to be re-evaluated. This selection is illustrated through the example given in section 4.2.1. After that, we show how the processor updates the target document by incrementally executing the transformation instructions.

5.1 Re-evaluation rules

After the source document has been modified, the incremental processor determines which instructions need potentially to be re-evaluated. This step relies on the preprocessing of *re-evaluation rules*: each re-evaluation rule consists of a pattern associated with a list of instructions to be re-evaluated. When a source node matches the pattern, then the list of associated instructions is likely to be re-executed. For instance, the instruction <xsl:apply-templates select="section"/> (line 18) will never be involved in the insertion of a title element. In contrary, when a section element is added, this expression can possibly be re-evaluated (depending on the depth level of the section). This basic selection of instructions to be re-evaluated can be enhanced. For instance, by taking into account that the expression is declared in a template that matches only article elements. In this case we can more accurately say if the apply-templates instruction needs to be re-evaluated. The apply-templates instruction will be executed only for nodes that match the article/section pattern.

The list of re-evaluation rules is built by considering all the expressions in the transformation sheet. Each expression is converted to a set of patterns and for each pattern a re-evaluation rule is created with the corresponding instruction.

In the next section, we describe how to implement a basic selection mechanism. Then we propose some optimizations based on context declaration, and variable de-referencing.

5.1.1 Basic selector

As said earlier, the creation of re-evaluation rules set from an expression can be quite complex. It consists of the identification of patterns that matches source nodes sensitive to the modification of the expression's result. These patterns must not contain dynamic *processor* context references, such as variable value and context size. In the following, we first consider the conversion of expressions without giving details of how location path conversion is achieved. Location path is introduced gradually. First, we describe location path conversion without considering predicates. Then, a more general approach including predicates is given. At the end of this section, we consider the case where template instantiation must be re-considered after a source modification.

5.1.1.1 Expression conversion

An expression is composed of operations, functions, variables and basic objects. The conversion of each of these components relies on the following informal algorithm:

- Operations such as and, or, equals generates patterns corresponding to those produced by the operation arguments. For example, the expression count(section) or position()>3 generates two patterns: section and node(). The number 3 does not generate patterns (see basic objects conversion rules).
- Functions:
  - Functions that use the dynamic context, such as position() or last() functions, must be re-evaluated when the current node list or the current node position changes. At this stage, as no additional information is provided, this kind of functions are replaced by the node() pattern. In section 5.1.3, we describe a basic optimization operation based on context-awareness.
  - Functions conversion with location paths as parameters produces re-evaluation rules corresponding to those of the location paths conversion. For example, count(section) expression is converted to section pattern.
  - The other functions do not generate any patterns. For example, the expression floor(43/2) never needs to be re-evaluated. Therefore, it does not generate patterns.
- A variable reference generates the node() pattern. It corresponds to the values the variable can take, in particular a node set (represented by node() pattern).
- Basic objects, such as number, string or boolean do not generates any patterns.

5.1.1.2 Location path without predicates conversion

The result of the location path is obtained by querying source nodes during each step of the path traversal. Consequently, as the result of the location paths depends on the results of the intermediate node sets, a pattern is generated for each of these intermediate node-sets. For example, the evaluation of artheader/authorgroup location path can change if an artheader element or an authorgroup element are added or removed in the source document. In this case, two re-evaluation rules are created for this particular expression.

In addition, the re-evaluation of an expression is not only required when the result node-set change. In fact, when expressions are converted to the string type, the evaluation of the expression is the list of text nodes descendants of the first node in the node-set. Therefore, a pattern is added in order to reflect a modification of these text nodes. For example, the expression title (line 16), as specified in the value-of instruction, is converted to a string type. So the conversion generates two patterns: title and title/descendant::text().

Taking into account axis relationship between steps can provide a basic optimization. In the previous example, the addition of an artheader element changes the path result only if this later contains an authorgroup element as a child. In the same manner, the addition of an authorgroup element changes the evaluation of the location path only if his parent is an artheader element.

5.1.1.3 Location path with predicates conversion

To identify if the evaluation of predicates can change after a source modification, they are converted in the same manner as described in the previous sections. The predicate evaluation is achieved in the context determined by the path step attached to the predicate. As a consequence, all the functions, which require the dynamic context (such as position()), can be converted as is. Only variable references must be converted.

5.1.2 Instantiation re-consideration

The execution of the apply-templates (and apply-imports) instruction, more than the nodes selection, must search for the template that best matches every selected node. This search operation (the instantiation process) may be reconsidered when a document source is modified. For instance, consider the following template definitions:

```
<xsl:template match="section[title]"> ... </xsl:template>
<xsl:template match="section"> ... </xsl:template>
```

The instantiation of a section element can change when a title element is added (or removed) to a section element. So an apply-templates instruction that has selected section nodes need potentially to be re-evaluated. In general, the instantiation process must consider all the templates defined in the transformation sheets. Therefore, each time the source document is modified, the entire previous instantiations must be reconsidered. To avoid this cost overhead, we take into account the fact that, most of the time, only some template rules can be instantiated for a given apply-templates instruction. From theses apply-templates/template dependencies, the instantiation processes related to a particular apply-templates instruction can be limited to a subset of the template rules. In summary, a re-evaluation rule is added for each template pattern that depends on an apply-templates instruction. Notice that the computation of apply-templates/template dependencies can also be used to enhance the instantiation processing performance during a batch transformation.

Formally, the problem of computing apply-templates/template dependencies can be formulated as following:

Let N_a be the node set that the apply-templates instruction a can select. Let N_t be the node set that the template rule t can instantiate. The template rule t is said to be dependent on the apply-templates a if and only if a node n belonging to N_a that belong to N_t exist. Formally:

$$T \text{ depends on } A \Leftrightarrow \exists\, n \in N_a / n \in N_t \Leftrightarrow A \text{ match } T$$

In practice, the way to determine such dependencies is to perform a pattern matching between the expression of the apply-templates instruction (and not the result of the expression) and template's pattern. Compared to classical tree pattern matching, the pattern matching is achieved here between two node-sets (represented by a pattern) and not between a node and a pattern.

In the following, we give some formulas in order to perform pattern matching between expressions and patterns. We do not present an exhaustive case study. We rather focus only on the child and attribute axes cases.

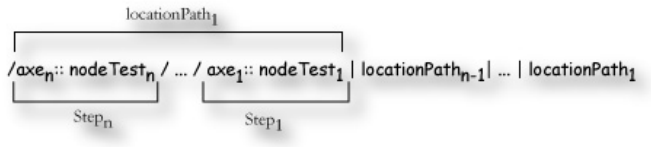

Figure 5. General syntax of path expressions (without predicates).

The following formulas are written using the expression general syntax of the apply-templates instruction, as illustrated in Figure 5.

An expression matches a pattern by respecting the following rules:

- Node test constructor:
  - $nodeTest_a$ match $node()_t \Leftrightarrow$ true
  - $nodeTest_a$ match $text()_t \Leftrightarrow \exists n \in nodeTest_a$ / type(n)=text()
  - $nodeTest_a$ match $nameTest_t \Leftrightarrow \exists n \in nodeTest_a$ / type(n)=element() $\wedge$ (name(n) = name($nameTest_t$) $\vee$ name($nameTest_t$) = '*' $\vee$ name(n) = '*')
- Axe constructor:
  - axe_a::$nodeTest_a$ match child::$nodeTest_t \Leftrightarrow$ $nodeTest_a$ match $nodeTest_t \wedge axe_a$ = child
  - axe_a::$nodeTest_a$ match attribute:$nodeTest_t \Leftrightarrow$ $nodeTest_a$ match $nodetest_t \wedge axe_a$ = attribute.
- Step constructor:
  - $step_{2a}$/ axe_a::$nodeTest_a$ match $step_{2t}$/axe_t::$nodeTest_t \Leftrightarrow$ axe_a::$nodeTest_a$ match axe_t::$nodeTest_t \wedge$ $step_{2a}$ match $step_{2t}$
- Union constructor:
  - $locationPath_{a2}$ | $locationPath_{a1}$ match $locationPath_{t1} \Leftrightarrow$ $locationPath_{a2}$ match $locationPath_{t2} \vee$ $locationPath_{a1}$ match $locationPath_{t1}$

In the previous sections, we have presented a basic selector that allows selecting the instructions to be re-evaluated for a particular node modification. This set of instructions is not minimal in that it contains instructions that are not necessary in the re-evaluation. In the following section, we will describe two techniques that we have introduced to minimize the number of these instructions.

5.1.3 Optimizations

5.1.3.1 Context awareness

The first optimization technique is the **context declaration awareness.** This optimization relies on the context of the instruction declaration. Three levels of this context are identified:

- **The context inside a predicate**: when converting a predicate, relative location paths inside the predicate are evaluated from the predicate's step. Such predicates are then prefixed by the left part of the predicate's step.
- **The context inside the template declaration**. Each relative pattern is evaluated on nodes that the template (or for-each instruction) has matched. Therefore, adding the template pattern as a prefix can refine relative patterns found during the basic selection. For instance, the section pattern associated to the apply-templates instruction line 18 can be transformed to article/section pattern.
- **The context outside the template declaration**. In some cases, a template can be instantiated by a limited number of apply-templates instructions (apply-templates/templates dependencies: see section 5.1.2). This information can be used to refine the re-evaluation pattern. Formally, let at_1, at_2, … at_n the apply-templates instructions that can instantiate the template t. Let p_1, p_2, … p_n the relative patterns of the template's instructions. Let pat_1, pat_2, …pat_n the (recursive) pattern conversion attached to at_1, at_2, … at_n apply-templates expressions. Therefore, p_1, p_2, … p_n patterns can be refined by the following patterns: (pat_1| pat_2 |… | pat_n)/p_1, (pat_1| pat_2 | … | pat_n)/p_2, … (pat_1| pat_2 |… | pat_n)/p_n. This expression of patterns must then be transformed to its canonical form. The canonical form allows the identification of equivalent patterns; which in turn allows removing redundancies from the pattern list.

In comparison with the basic selection, all the patterns obtained now are more precise. Furthermore, they have generally less associated instructions to be re-evaluated. For instance, in the basic selection, when the title of a section is modified, eight instructions need to be re-evaluated (lines 18, 24, 51, 57, 69, 17, 48 and 53). While with the enhanced selection, we have just five that need to be re-evaluated. Moreover, this selection is able to distinguish between a section inserted in another section and a section inserted in an article.

5.1.3.2 Variables

The second optimization is related to **variables.** In some cases, the value of variables is just limited to a subset of values. In the example given in the annex A (line 36), the value of the variable named heading depends only on the number of section ancestors of another section. So the value of the attribute named name (line 45) need to be re-evaluated only if a section element is added (or removed) in the source document. By de-referencing the variables that support de-referencing (with no reference to a parameter), it's then possible to refine the pattern obtained by a conversion of an expression with a variable. A more general solution that covers parameter references is under consideration, in particular, by using template/apply-templates dependencies.

5.1.4 Synthesis

In the example given in section 4.2.1, we gave the example of a modification of the source document by inserting a new section element to the article element. The result of the selective re-computation, obtained by applying the enhanced selector, corresponds to the instructions at lines 18, 25, 45, 48, and 51. Compared to the instructions that ideally needed to be re-computed (see section 4.2.1), two additional instructions need to be re-evaluated. In fact, the instructions at lines 45 and 48 are re-evaluated because they contain a variable reference. If we consider that we had 18 instructions that compose the transformation sheet, our selection method reduced the size of the instructions under consideration to five instructions.

In general, obtaining an efficient incremental transformation depends on the manner the transformation sheet has been written. For example, if the transformation sheet programmer limits the use of wildcards in expressions, the performance of the incremental processor will increase substantially. This is due to a more accurate selection. In general, the transformation sheet design affects not only the incremental transformation but also the batch transformation.

As the list of instructions to be re-evaluated is known beforehand, some hints on the incremental transformation processing cost can be obtained. For example, if the instruction list to be re-evaluated is short and if it does not contain instructions of flow type, the re-evaluation is likely to be very fast. Having performance hints is very valuable in an authoring environment. For instance, when a character is typed on the target, a high priority can be set on instructions that update the corresponding target (instruction with a lower cost). The slower instructions that allow propagating this change in the document can be executed with a lower priority.

5.2 Incremental execution

5.2.1 Basic process

Thanks to the list of re-evaluation rules obtained using the selection introduced in the previous section, the instruction to be re-evaluated can now be determined. This is achieved by applying a pattern matching between re-evaluation rule patterns and the node currently being modified. Instructions associated to a pattern that matches that node need to be re-evaluated.

The processing of these instructions is achieved by performing a depth first traversal of the execution flow. When an execution node is about to be traversed, depending on the execution node type, a specific action is executed:

- **Flow.** If the instruction needs to be re-evaluated, then it is executed incrementally as described in the algorithm below for the apply-templates instruction.
- **Producer.** If the instruction needs to be re-evaluated, then it is executed. Otherwise, the target context is only updated. For character producer instructions, the number of previously generated characters is increased. For element producer instructions, the number of characters is set to zero and the associated target node is set to be the current one.
- **Variable.** The variable declaration is pushed on a variable stack. Its value is not computed now but later when needed.
- **Parameter.** The parameter declaration is also pushed on the variable stack and its value computed when needed.

At the end of the traversal of an execution node, similar actions are performed at the node level: variables declarations are popped from the stack, etc.

When an instruction is executed, the processor starts by computing the value of variables needed by the expression. Then, depending on the instruction, an incremental algorithm is executed. Figure 6 gives the algorithm that allows the incremental execution of the apply-templates instruction.

```
nodeList <- Evaluate select expression
If ( sort instruction specified )
  sort( nodeList )
End If
For each Node in NodeList
  If ( node ∉ previousNodeList )
    // The source node was not selected during the previous transformation
    Execute apply-templates instruction with Node as context
  Else
    // Test if the template matching has changed
    template <- findTemplate( node )
    If ( template = previousTemplate( node ) )
      // Same template
      If ( not same position as previously )
        // The generation order has changed
        changeTargetPosition()
      End If
      Incrementally execute children with Node as context
    Else
      // Not the same template
      Destroy previously generated target
      Execute apply-templates instruction with Node as context
    End If
  End
End For
```

Figure 6. The incremental algorithm for the apply-templates instruction

The algorithm for the for-each instruction is quite similar, except that there is no template matching needed. As a consequence, the test that checks if template matching has changed is not used. On the contrary, for the apply-imports instruction the algorithm contains only that test. For the value-of instruction, the new characters replace the previously generated ones.

As the target document is traversed in parallel to the execution flow tree, producer execution nodes don't need to be stored. Therefore, memory consumption is substantially reduced (see evaluations in section 6).

5.2.2 Optimizations

In this section, we present two optimizations that can improve the performance of the incremental processor.

The first one is the source nodes selection. When an instruction is to be re-executed, it is re-executed for all instantiated source nodes. For template rules that are instantiated frequently, the number of re-executions can be more than necessary. In some cases, it is possible to prevent some superfluous executions: for instance, the instruction line 53 need to be re-evaluated only for parent node of the editing node and not for all nodes instantiated by the corresponding rule.

To achieve this selection, depending on what triggered the instruction re-evaluation, the selection will be achieved in two manners. First, the result node set of the location paths that composed the expression has changed. In this case, the list of source nodes can be easily obtained in the same manner as the expression evaluation. Starting from the editing node, the location path expression is executed in the reverse order (from right to left) until the end of path. The result is a set of source nodes. For example, the instruction line 56 needs to be re-evaluated when a section element is added or removed. Source nodes for which this instruction must be re-evaluated is given by the evaluation of the reverse expression section/descendant::section. The context node for evaluating this expression is the edited section element. Obviously, this optimization is possible only if the expression does not contain dynamic references.

Second, the list of nodes in the context (context node list) has changed. It must be restored (in the same manner as incremental execution of apply-templates instruction). If the edited node appears in the context node list (or do not appear anymore), then the expression must be re-executed for the entire source nodes contained in the context node list. As the context node list cannot change without an execution of apply-templates and for-each instruction, computing the list of source nodes can be achieved during the execution of these instructions.

The second optimization is the pattern inclusion. In order to identify the list of instructions to re-evaluate, a pattern matching is

performed on the entire re-evaluation rules set. To avoid such an overhead, each time a pattern is converted from an expression, the expression's instruction is added to the re-evaluation rules that include the new pattern. A pattern P1 includes another pattern P2 when all nodes that matches pattern P2 match pattern P1. So, in order to find the list of instructions to re-evaluate, it is sufficient to find the first pattern that matches. Algorithms described in section 5.1.2 can be used to perform pattern matching between patterns.

The incremental transformation with these optimizations allows updating, in a reasonably responsive application, the target documents after a change in the source document.

6. EVALUATION

The techniques, presented earlier, allow the implementation of incremental transformation processors. They have been partially integrated in the Xalan batch processor from the Apache Foundation Software [3]. The basic selector, presented in section 5.1.1, has been almost completely implemented. In the current version of our application, it supports a subset but significant part of the expression constructors. The context awareness inside the template declaration optimization, presented in section 5.1.3, has been implemented. An extended pattern matching that takes into account the new pattern definition, introduced in section 4.2.3, has also been implemented. The incremental execution process traverses the execution tree. Variables and parameters are evaluated during this tree traversal.

In order to evaluate our current implementation of the incremental processor, some measures of the costs in terms of speed and memory space requirement have been achieved. They are summarized in Table 1 and Table 2.

Table 1. Speed of the transformations applied to Norman Walsh's docbook transformations sheet.

	Batch		Dummy		Change article title		Insert section	
Number of instructions to re-execute	N/A		0		795		819	
Time to get instruction to re-evaluate	N/A		0		80ms		80ms	
Variables computed	6572		6572		6572		6572	
Variable access count	10279		6899		6899		6983	
Overall timing/ratio	4,5s	1	2,8s	0.62	2,8s	0.62	2,9s	0.64

These tables show that incremental transformations are very promising in terms of the overall system's performance. For speed consideration, we have measured some particular operations such as appending an author in the working example. The result is that most of the changes remain within a reasonable processing time. This is very important since incremental processors are used in an interactive mode and have to remain responsive. The memory usage is not prohibitive too. The overhead that we measured, that is mainly due to the size of the execution flow tree, represents a small fraction of the memory occupied by the source and target documents. We have applied our measures to two docbook documents, using Norman Walsh's transformation sheets [21] to produce HTML documents. These transformation sheets contain 1236 templates and are not optimized for incremental processing. It contains a lot of generic expressions and variables. Consequently, the number of instructions to re-evaluate is widely over-estimated. And, a lot of processing time is wasted because of systematic variable values computation during tree traversal, which represent close to 100 percent of incremental process. We expect that the implementation of optimizations such as template context awareness and variable de-referencing (see section 5.1.3) will strongly improve incremental performance for such transformation sheets.

Table 2. Speed of the transformations applied to extended version of transformation sheet in Annex A.

	Batch		Dummy		Add an author		Insert section	
Number of instructions to re-execute	N/A		0		18		20	
Time to get instruction to re-evaluate	N/A		0		<10ms		<10ms	
Variables computed	133		133		133		133	
Variable access count	1563		0		25		36	
Overall timing/ratio	2,3s	1	0,1s	0.04	0,2s	0.08	0,3s	0.13

Table 3. Memory size of the transformations (Norman Walsh's docbook transformations sheet).

	Normal document	High document
Source Document (Kbytes)	193	505
Target document (Kbytes)	224	596
Execution flow tree (without target) (Kbytes)	106	259
Execution flow tree (with target) (Kbytes)	153	364
Ratio (source + target) / execution	36%	33%

A second evaluation has been performed on a little transformation sheet (50 templates) applied to the same documents. This transformation sheet, which is an extension of the transformation sheet given in annex A, has been written using few variables and no wild cards. Consequently, instructions that need to be re-evaluated are determined more accurately than Norman Walsh's transformation sheets.

7. CONCLUSION AND PERSPECTIVES

In this paper, we have presented an incremental transformation framework called *incXSLT*. This framework has been experimented for the XSLT language from the World Wide Web Consortium. XSLT has gained a very large acceptance and is currently deployed in several systems such as Web browsers,

publishing tools and database systems. In these systems, the XML content together with the transformation sheets are designed in a batch mode. Transformation sheets designers usually debug their sheets using batch processors. This operation is tedious since it does not allow identifying precisely the error sources. In addition, it prevents the identification of the immediate effect of source document or transformation sheet changes in the destination document. Incremental transformation processors such as *incXSLT* represent a better alternative for the design of both the content and the transformation sheets. It is a first step toward full blown interactive transformation-based authoring environments.

In the short term, we plan to extend the implementation to support most of the selector optimizations described in section 5.1. The goal is to cover incremental transformations closer to real world source documents and transformation sheets. The second goal is to allow a more flexible editing of the transformation sheets. For a lack of space, we have not detailed this aspect in the paper. Nevertheless, the execution flow tree data-structure remains the central vehicle of the transformation updates.

In the future, we expect that the specification of transformation sheets will remain a difficult task. We think that, in the longer term, it could be useful to identify common transformation schemes in order to make them available as building blocks. Ideally, making the composition of such building blocks available through a GUI interface will help greatly in making transformations accessible to a wider range of users. A second goal that we need to address is the problem of reverse transformations. The reverse transformation problem can be formulated as follows: for a given target modification, the problem is to find all the modifications to apply on the source document to obtain that particular modification. This allows suggesting to a document author the locations in the source document where to apply a change in order to have that target modification.

8. REFERENCES

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "An execution-Backtracking Approach to Debugging", IEEE Software, pp. 21-26, May 1991.

[2] Altova, "XML Spy 3.0", http://www.xmlspy.com/, 2000.

[3] Apache XML project, "Xalan", 2000. http://xml.apache.org/xalan/index.html

[4] Balise, "Balise 4", 2000. http://www.us.balise.com/products/balise/index.htm

[5] "Composite Capabilities/Preference Profiles: Requirements and Architecture", M. Nilsson, J. Hjelm and H. Ohto, W3C Working Draft, available at http://www.w3.org/TR/CCPP-ra/, 21 July 2000.

[6] "Document Object Model (DOM) Level 2 Views Specification", W3C Recommendation, available at http://www.w3.org/TR/DOM-Level-2-Views/, 13 November 2000.

[7] "Extensible Stylesheet Language (XSL) Version 1.0", S. Adler and Co, W3C Working Draft, available at http://www.w3.org/TR/xsl/, 18 October 2000.

[8] C. Hoffmann and M. O'Donnell, "Pattern Matching in Trees", Journal of the Association for Computing Machinery, vol. 29, n°1, pp 68-95, January 1982.

[9] M. Kay, "XSLT Programmer's Reference", Wrox Press, 2000.

[10] Y. A. Liu, "Efficiency by incrementalization: An introduction", Higher-Order and Symbolic Computation, 13(4), 2000.

[11] R. H. B. Netzer and M H. Weaver, "Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs", In Proceedings of the ACM SIGPLAN '94 conference on Programming language design and implementation, 1994.

[12] "Namespaces in XML", T. Bray, D. Hollander, A. Layman. W3C Recommendation, available at http://www.w3.org/TR/REC-xml-names, 14 January 1999

[13] Oasis, "Docbook", http://www.docbook.org.

[14] Omnimark, "Guide to OmniMark 5", 2000. http://www.omnimark.com/develop/om5/doc/

[15] V. Quint, C. Roisin and I. Vatton, "A Structured Authoring Environment for the World-Wide Web", Proceedings of the Third International World-Wide Web Conference, Computer Networks and ISDN Systems, vol. 27, num. 6, pp. 831-840, April 1995.

[16] G. Ramalingam and T. Reps, "A categorized bibliography on incremental computation", In Conference Record of 20th Annual ACM Symposium on Principles of Programming Languages, pages 502-510, ACM, New York, Jan 1993.

[17] T. Reps, T. Teitelbaum and A. Demers, "Incremental context-dependent analysis for language-based editors", ACM Trans. Program. Languages System, vol. 5, num 3, pp. 449-477, July 1983.

[18] SoftQuad, "XMetal 2.0", 2000. http://www.xmetal.com/

[19] Vervet Logic, "XML Pro v2", 2000. http://www.vervet.com/

[20] L. Villard, C. Roisin and N. Layaïda, "A XML-based multimedia document processing model for content adaptation", In Proceeding of Digital Documents and Electronic Publishing (DDEP00), 2000.

[21] N. Walsh, "XSL DocBook Stylesheets", http://nwalsh.com/docbook/xsl/index.html, 30 January 2001.

[22] WhiteHill, *"<xsl>Composer"*, 2001. http://www.whitehill.com/

[23] "XML Path Language (XPath)", J. Clark and S. DeRose, W3C Recommendation, available http://www.w3.org/TR/xpath.html, 16 November 1999.

[24] "XSL Transformations (XSLT)", J. Clark, W3C Recommendation, available at http://www.w3.org/TR/xslt, 16 November 1999.

ANNEX A

```
1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.
3.  <xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4.    <xsl:output method="html"/>
5.
6.    <xsl:param name="toc.depth">2</xsl:param>
7.
8.    <xsl:template match="article">
9.      <html>
10.       <body>
11.         <p align="center">
12.           <xsl:value-of
    select="artheader/authorgroup/author/firstname"/>
13.           <xsl:text> </xsl:text>
14.           <xsl:value-of
    select="artheader/authorgroup/author/lastname"/>
15.           <xsl:text>Presents</xsl:text>
16.         </p>
17.         <h1 align="center"><xsl:value-of
    select="title"/></h1>
18.         <xsl:apply-templates select="section">
19.           <xsl:with-param name="indent">0</xsl:with-
    param>
20.         </xsl:apply-templates>
21.         <hr/>
22.         <table border="1" width="100%">
23.           <tr>
24.             <td><xsl:apply-templates
    select="artheader/authorgroup"/></td>
25.             <td>Nb upper sections : <xsl:value-of
    select="count(section)"/></td>
26.             <td>Last modification : <xsl:value-of
    select="artheader/date"/></td>
27.           </tr>
28.         </table>
29.       </body>
30.     </html>
31.   </xsl:template>
32.
33.   <xsl:template match="section">
34.     <xsl:param name="indent">0</xsl:param>
35.
36.     <xsl:variable name="heading">
37.       <xsl:choose>
38.         <xsl:when test="count(ancestor::section) =
    0">h2</xsl:when>
39.         <xsl:when test="count(ancestor::section) =
    1">h3</xsl:when>
40.         <xsl:when test="count(ancestor::section) =
    2">h4</xsl:when>
41.         <xsl:otherwise>p</xsl:otherwise>
42.       </xsl:choose>
43.     </xsl:variable>
44.
45.     <xsl:element name="{$heading}">
46.       <xsl:attribute name="align">left</xsl:attribute>
47.       <xsl:attribute name="style">padding-left=
48.         <xsl:value-of select="$indent"/>px
49.       </xsl:attribute>
50.
51.       <xsl:number value="position()" format="1."/>
52.       <xsl:text> </xsl:text>
53.       <xsl:value-of select="title"/>
54.     </xsl:element>
55.
56.     <xsl:if test="count(ancestor::section) &lt; $toc.depth
    - 1">
57.       <xsl:apply-templates select="section">
58.         <xsl:with-param name="number-
    format">a.</xsl:with-param>
59.         <xsl:with-param name="indent" select="$indent
    + 100"/>
60.       </xsl:apply-templates>
61.     </xsl:if>
62.   </xsl:template>
63.
64.   <xsl:template match="authorgroup">
65.     <xsl:for-each select="author">
66.       <xsl:value-of select="firstname"/>
67.       <xsl:text> </xsl:text>
68.       <xsl:value-of select="lastname"/>
69.       <xsl:choose>
70.         <xsl:when test="position()=last()-1"><xsl:text>
    and </xsl:text></xsl:when>
71.         <xsl:when test="position() &lt; last()-
    1"><xsl:text>, </xsl:text></xsl:when>
72.         <xsl:otherwise/>
73.       </xsl:choose>
74.     </xsl:for-each>
75.   </xsl:template>
76.
77. </xsl:stylesheet>
```