XML transformation flow processing

Jérôme Euzenat
INRIA Rhône-Alpes
Montbonnot, France
web http://www.inrialpes.fr/exmo
email Jerome.Euzenat@inrialpes.fr

Laurent Tardif
Fluxmedia
Montbonnot, France
web http://www.fluxmedia.fr
email laurent.tardif@csse.monash.edu.au

ABSTRACT

The XSLT language is both complex in simple cases (such as attribute renaming or element hiding) and restricted in complex cases (complex information flows require processing multiple stylesheets). We propose a framework which improves on XSLT by providing simple-to-use and easy-to-analyse macros for common basic transformation tasks. It also provides a superstructure for composing multiple stylesheets, with multiple input and output documents, in ways not accessible within XSLT. Having the whole transformation description in an integrated format allows better control and analysis of the complete transformation.

KEYWORDS

XML; XSLT; Transmorpher; Transformations

1 INTRODUCTION AND MOTIVATION

In electronic documentation, the notion of a transformation is widespread. It is a process that transforms a source document into another document, the target. This notion concerns the whole computing discipline with the advent of the XML language.

As there are multiple computing practices, there are multiple needs for a transformation system. We motivate and present here a system that targets increased intelligibility in the expression of transformations. This need is first motivated before telling why, in our opinion, XSLT falls short of the objectives of simplicity and power. The requirements for such a system are then presented.

1.1 Motivating example

Consider someone wanting to generate part of a web site concerning bibliographic data. The source of information is a set of XML formatted bibliography documents, containing `reference` elements described by `authors`, `title` or `abstract` elements. The system aims at providing several different documents:

- An HTML presentation ordered by types and year of publication,
- A list of bibtex entries sorted by category, authors and year,
- An HTML rendered XML presentation of the bibliography, and
- The XML file of references for two particular authors.

The two first documents must have first been stripped of abstract and non-public information.

The generation of the first three documents can be naturally expressed by the following schema in which boxes are transformations written in some transformation language (e.g. XSLT) and `strip-abstract` is the simple suppression of abstract elements, of elements marked as private and of mark attributes.

The picture represents what we call a transformation flow, i.e. a set of transformations linked by information channels. It is worth noting, that a transformation flow does not indicate if it must be processed in a demand-driven (pull) or data-driven (push) manner.

1.2 Limitations of XSLT

XSLT [Clark 1999a] is a very powerful technology for transforming XML documents which has been carefully designed for rendering. It has the advantage of being based on XML itself. Of course, all the manipulations that have been described in the example can be expressed in XSLT. Yet, XSLT suffers from a few shortcomings that make it both too sophisticated and too restricted at once. These shortcomings are:

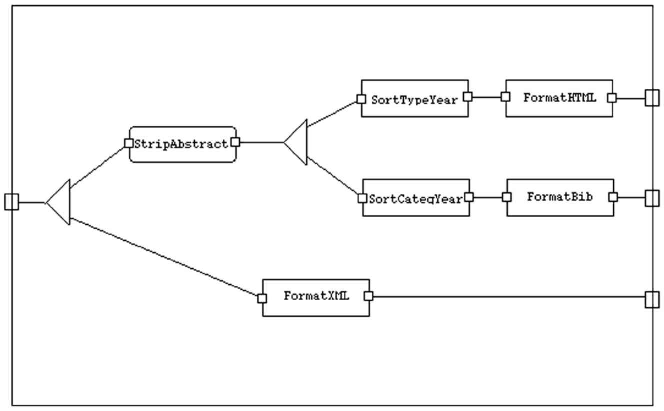

Figure 1 A sample transformation flow (flowing from left to right)

Complexity Writing simple transformations (tag translation, tree decoration, information hiding such as the abstract stripping in the example) requires knowledge of XSLT even though they can be expressed in a straightforward manner by the user. There is no simple way to implement these transformations in XSLT.

Lack of intelligibility If it is easy to parse an XSLT stylesheet, it is not easy to understand it because roughly the same construct with many different parameters is used for writing both simple transformations and sophisticated ones. To XPath [Clark 1999b], XSLT [Clark 1999a] adds seven extension functions, including `document()` for processing multiple documents. The treatment of multi-source transformations concurs to XSLT's lack of intelligibility. through this `document()` construct, and the strong inequality of treatment with output specification that is dealt with through an explicit `output` XSLT construct (although, this could change a bit with XSLT 1.1 [Clark 2001a] it is not in a totally satisfying way).

As a consequence, building analysis tools is rather complicated. This is one of our deeper motivations. Analyzing transformation flows can be used for many purposes from displaying them (as in the above picture) to assessing the properties preserved by some transformations (and going towards proof-carrying transformations) and optimizing transformations.

Non self-sufficiency Writing complex transformation flows involving independently designed stylesheets, multi-document dataflow and closure operation requires the use of an external environment (scripting language, shell) and compromises portability of the transformation flows.

Other pieces of work have already addressed this issue: namely the AxKit and Cocoon projects which implement pipelining of stylesheets. However, since they are concerned with demand-driven documents, they do not address the multi-output and complex dataflow issue (i.e.

when a document generates several outputs that are themselves subject to independent transformations and can eventually be merged later on). These issues are important for the future XML-based information systems.

Limited power XSLT is not so limited as it may appear. But it has been designed in such a way that some powerful operations are difficult to process. A good example is the closure operation (i.e. applying a stylesheet until its application does not change the document anymore). Such an operation is very powerful and can be written very concisely. It can be used for gathering all the nodes of a particular graph (e.g. flattening a complete web site into one document can be seen as a closure operation).

For those who need these operations, they can either implement them outside XSLT (in a non portable shell) or inside XSLT (with extra contortions).

This limited power issue is sometimes described as a lack of side effects. However, XSLT provides side-effects by applying a transformation to a document and then reading that document through an XSLT specific XPath construct `document()` call. Moreover, recursive expressions can be written inside XSLT as shown by [Kay 2000a][Becker 2000a].

In order to overcome these problems, we have started to design a system that relies on XSLT and attempts to remain compatible with it but embeds it in a superstructure. This system, called Transmorpher, is the subject of this paper.

1.3 Requirements

Transmorpher is an environment for processing generic transformations on XML documents. It aims at complementing XSLT in order to:

- describe simple transformations easily (removing elements, replacing element and attribute names, concatenating documents...);
- allowing regular expression transformations on the content;
- composing transformations by linking their (multiple) output to input;
- iterating transformations, sometimes until saturation (closure operation);
- integrating external transformations.

The guidelines of the proposal are the following:

- being as compatible as possible with XSLT (by defining transformations from many Transmorpher elements to XSLT);
- being portable: it is implemented in Java (as opposed to generating XSLT stylesheet and scripts which would have been an easy solution);
- being open to other systems by importing any kind of stylesheet and expressing control over it;
- self-contained transformation features.

In the remainder of this paper, the design of Transmorpher is presented. The next section presents its computing model involving the composition of transformations. Then, the built-in abstract basic transformations which can be handled by Transmorpher are presented. The notion of rules for expressing straightforward transformations in a drastic simplification of XSLT is detailed. We end with a quick description of the current implementation and a comparison with other work.

2 COMPUTING MODEL

Transformation flows are made of sets of transformations connected by channels on their input/output ports. Transformations can in turn be either transformation flows or elementary transformations. Channels carry the information to be transformed (currently, only XML-formatted under the form of SAX events [Boag 2000a]). They can take several inputs and provide several outputs during one execution. The Transmorpher computing model is thus rather simple.

Transmorpher enables the description transformation flows in XML. It also defines a set of abstract elementary transformations that are provided with an interface and execution model. Currently, the available transformations are: generators, serializers, rule set processors, dispatchers, mergers, query evaluators, external processing calls and iterators.

The interpretation of a transformation flow consists of creating the transformations, connecting them through channels and providing input to the source input channels. This interpretation can be triggered at the shell level, or programmed in another application and we are working towards making it usable as a servlet.

Transmorpher is thus made of two main parts: a set of documented Java classes (which can be refined and integrated in other software) and an interpreter of transformation flows. The transformation flows can be specified by programming the class instantiation in Java or by describing it in XML.

2.1 Processes

The transformation flows are described in an XML document which clearly separates the rules from the processing. The transformation flows are described through a `process` element. There can be several such processes in one document.

The processes contain a set of subprocesses, whose main types are:

`<apply-process name='name'/>` which calls an already defined process,

`<apply-ruleset name='name' strategy='strategy' />` which applies a set of rules (equivalent to an XSLT stylesheet) to its input,

<apply-external type='type' file='file' /> which calls an external procedure on the input and must provide the output. This engine could be Perl, XSLT, or whatever is appropriate.

<apply-query name='name' type='type' file='file' /> which evaluates a query on the input and must provide the output. This query engine could be XQL, SQL, or whatever is appropriate.

<repeat times='n' buffer='channels'/> which applies the contained treatment a particular number of times (or until the input and output are the same). Buffering channels are provided for expressing the information flow.;

Other instrumental subprocesses are:

<dispatch type='type'/> which takes one input and several outputs,

<merge type='type'/> which takes several input and one output,

<generate type='type'/> which takes no input and one output (generally used to read from outer streams like files),

<serialize type='type'/> which takes one input and no output (generally used to write to outer streams like files).

Each of these primitives has an id (enabling the identification of subprocesses of the same kind) and in and out attributes (enabling their connection to other processes). The name attribute denotes an element defined within the current transformation (or an imported one). The type attribute identifies a particular implementation of the basic process.

Adding other basic processes to Transmorpher should be simple because they are simply other elements to add to the DTD.

Below is a Transmorpher transformation flow, in which the processGeneral process corresponds to the flow described by the figure above.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE transmorpher SYSTEM "../../dtd/transmorpher.dtd">

<transmorpher name="generateBiblio"
     xmlns="http://transmorpher.fluxmedia.fr/1.0"
     xmlns:regexp="xalan://
           fr.fluxmedia.transmorpher.regexp.RegularExpression">

  <ruleset name="stripAbstract">
    <remtag match="abstract" context="reference"/>
    <remtag match="keywords" context="reference"/>
    <remtag match="areas" context="reference"/>
    <remtag match="softwares" context="reference"/>
    <remtag match="contracts" context="reference"/>
    <remtag match="*[@status='hidden']"/>
    <resubst match="conference/@issue" source="([0-9]+)"
         target="$1e"/>
    <rematt match="status"/>
    <rematt match="isbn" context="book"/>
  </ruleset>

  <query name="troncybrunet" type="tmq" root="bibliography">
    <select match="bibliography/reference[authors/p/
                                                @last='Troncy']"/>
    <select match="bibliography/reference[authors/p/
                                                @last='Brunet']"/>
  </query>

  <process name="processGeneral" in="R112" out="X3 Y3 Z3">
     <dispatch type="broadcast" id="dispatch 2" in="R112"
                                                    out="D1 D3"/>

     <apply-ruleset ref="stripAbstract" id="StripAbstract"
in="D1" out="X1"/>
     <dispatch type="broadcast" id="dispatchStripped" in="X1"
                                                  out="X11 X12"/>

     <apply-external type="xslt" id="SortTypeYear"
                                      file="biblio/sort-ty.xsl"
         in="X11" out="X2"/>
     <apply-external type="xslt" id="FormatHTML" file="biblio/
                                                  form-harea.xsl"
         in="X2" out="X3"/>

     <apply-external type="xslt" id="SortCategYear"
                                     file="biblio/sort-cya.xsl"
         in="X12" out="Y2"/>
     <apply-external type="xslt" id="FormatBib" file="biblio/
                                                 form-bibtex.xsl"
         in="Y2" out="Y3"/>

     <apply-external type="xslt" id="FormatXML" file="biblio/
                                        xmlverbatimwrapper.xsl"
         in="D3" out="Z3"/>
  </process>


  <process name="processByNames" in="R111" out="Z34">
     <apply-query type="tmq" ref="troncybrunet" id="FilterTB"
         in="R111" out="Z34" />
  </process>

  <main name="ProcessBiblio">
    <generate type="readfile" id="bibexmo" out="R1"
                                   file="biblio/bibexmo.xml"/>
    <generate type="readfile" id="je" out="R2" file="biblio/
                                                        je.xml"/>
    <merge type="concat" id="merge" in="R1 R2" out="R3"/>
    <dispatch type="broadcast" id="dispatch1" in="R3" out="X11
                                                          X12"/>

    <apply-process id="generateFormat" ref="processGeneral"
                                                      in="X11"
        out="X31 Y31 Z31" />
    <serialize type="writefile" id="writeHTML" in="X31"
                                  file="biblio/biblio.html" />
    <serialize type="writefile" id="writeBIB" in="Y31"
                                  file="biblio/biblio.bib"/>
    <serialize type="writefile" id="writeXML" in="Z31"
                               file="biblio/biblio-xml.html"/>

    <apply-process id="processTB" ref="processByNames" in="X12"
                                                  out="W3" />
    <serialize type="writefile" id="writeTB" in="W3"
                                file="biblio/biblio_tb.html"/>
  </main>

</transmorpher>
```

2.2 Channels

In Transmorpher the generic processes can have several input and several output port. These ports are connected to channels that are fed in by the output of a process and can be used as input of other processes. They are abstractions that enables the expression of the flow of information in a compound transformation and not the mark of a particular implementation. The set of channels is called the dataflow.

The channels specify a unit in which processes can read and write. They are named streams which can be visible from outside a process if they are declared as their input or output.

The control inside a process can be deduced from the dataflow. There is no explicit operator for parallelizing or composing transformations: their channels denote composition, precedence and independence of processes.

Alternative solutions to channels, could have been retained in order to deal uniformly with input/output. The solution taken by [Drewes 2000a] consists of considering each transformation as a function from one (not necessarily connected) graph to another. This solution has the advantage of using functions and sticking to the initial XSLT design but it does not preserve the order between the graphs. It can be replaced by the *nodeset* notion of XSLT/XPath. However, on the application side, the objects to be manipulated are documents and not graphs or node sets, so we kept on modeling multiple input/output transformations.

The channels are currently implemented by SAX2 event flows, with all data being encoded in UTF-8. They thus can only carry XML data (which can be text). Very few verifications are done so far on the channels, but it is possible to declare their DTD and statically check that the declarations coincide.

3 BUILT-IN ABSTRACT TRANSFORMATIONS

Transmorpher offers a set of abstract transformations which correspond to a Java interface specifying the number of input and output channels, the other attributes plus an expected behavior. These abstract transformations can be used in processes once they have been implemented.

We present below the set of these abstract transformations and the implementations currently provided by Transmorpher.

3.1 Generator and serializers

When a transformation flow takes input from a source which is not a stream (file, network, database...) or must output some information directly to such a location, there is a need for basic "transformations" achieving this simple task. This is useful when generating a full web site from source elements.

Generators (and serializers) are the most basic built-in components. They are transformations with no input and one output (and one input and no output respectively).

In the bibliographic example, the transformation flows always get their data from files bibexmo.xml and je.xml. These files are read through the use of the generate element:

```
<generate type="readfile" id="bibexmo" out="R1" file="biblio/
                                               bibexmo.xml"/>
<generate type="readfile" id="je" out="R2" file="biblio/
                                                    je.xml"/>
```

and the result is written to the files biblio.html, biblio.bib, biblio-xml.html and tb-biblio.html via the serialize elements:

```
  <serialize type="writefile" id="writeHTML" in="X31"
     file="biblio/biblio.html" />
  <serialize type="writefile" id="writeBIB" in="Y31"
     file="biblio/biblio.bib"/>
  <serialize type="writefile" id="writeXML" in="Z31"
     file="biblio/biblio-xml.html"/>
  <serialize type="writefile" id="writeTB" in="W3"
     file="biblio/biblio_tb.html"/>
```

Serializing enables, for instance, reading a file from the file system or the Internet. It can also generate XML from a database.

3.2 Dispatchers and mergers

It is handy to be able to send the same input to several processes. It is also useful to be able to gather several process outputs to one output channel. In order to do so in a simple manner, Transmorpher provides the basic dispatcher and merger processes.

3.2.1 Dispatcher

The dispatcher is an element that has one input and many outputs.

Transmorpher provides a basic implementation of a dispatcher, its behavior is to copy the input to the different outputs. It is used in the bibliography example for dispatching the input:

```
   <dispatch type="broadcast" id="dispatch 2" in="R112"
      out="D1 D3"/>
   <dispatch type="broadcast" id="dispatchStripped" in="X1"
      out="X11 X12"/>
```

One can imagine more complex dispatchers. In the bibliography example, a dispatcher could split the reference elements into a regular list of references and the list of authors. This can be useful for generating a list of authors (sorted and uniq'ed) or an index. In the transformation of this paper from XML to HTML, we could have separated the graphic elements from the rest of this paper in order to transform the TIFF pictures into JPEG.

3.2.2 Merger

The merger is a process that has several inputs and one output.

Transmorpher provides a simple notion of merger that copies all the 2..n channels under the root element of the first channel (preserving order). This merger is used in the bibliography example for merging two source XML documents:

```
<merge type="concat" id="merge" in="R1 R2" out="R3"/>
```

One can imagine more complex mergers. For instance, in the bibliography example, one can define a merger which merges a channel of titles with a channel of authors to produce a channel of references.

3.3 Query

One important requirement of some transformation applications is the ability to select some part of the XML documents to be further transformed. The query abstract transformation covers this goal.

It takes as input the source to be queried and the query to be processed against the input. Its output will be the result.

In order to test this interface we defined a very simple query language that only allows the expression of two elements: query which contains a set of select elements. The select elements have a match attribute containing an XPath [Clark 1999b] expression. The evaluation of the query will return all the subtrees of the input stream that match one of the select clauses. The conjunction can be implemented by composing two such queries. This is a very light query language which does not transform order or provide complex operations but it is embedded in a rich environment which can provide this.

For instance, the following query asks for all the references involving either Troncy or Brunet as author (in any position).

```
<query name="troncybrunet" type="tmq" root="bibliography">
  <select match="bibliography/reference[authors/p/
                                        @last='Troncy']"/>
  <select match="bibliography/reference[authors/p/
                                        @last='Brunet']"/>
</query>
```

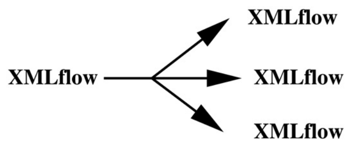

Figure 2 Basic dispatcher

The root attribute specifies that the result is returned within a bibliography element.

This simple query language has been implemented by transforming the query element into an XSLT stylesheet that is processed against the input and returns the result.

3.4 External processes

External processes allow users to take advantage of a transformation program that is not specific to Transmorpher. This can be very useful when some legacy programs exists, when one can implement more easily some transformations in another language or when Transmorpher is too limited for expressing the required transformation.

An external process can have several inputs and outputs, and is identified by its type. It can also take parameters under the form of a file (this is too limiting however and, in the future, we will introduce true parameters).

Transmorpher provides an XSLT external process type which enables the processing of XSLT stylesheets already defined (this was the case of the stylesheets of the bibliography example which have not been changed to run under Transmorpher). Of course, the XSLT external processes have only one input and one output. Our implementation takes advantage of Xalan that is embedded in Transmorpher, but it should be wrapped in JAXP for better portability.

In the bibliography example, the external XSLT processes are used for sorting and formatting:

```
<apply-external type="xslt" id="SortTypeYear"
                                  file="biblio/sort-ty.xsl"
   in="X11" out="X2"/>
<apply-external type="xslt" id="FormatHTML" file="biblio/
                                            form-harea.xsl"
   in="X2" out="X3"/>

<apply-external type="xslt" id="SortCategYear"
                                 file="biblio/sort-cya.xsl"
   in="X12" out="Y2"/>
<apply-external type="xslt" id="FormatBib" file="biblio/
                                           form-bibtex.xsl"
   in="Y2" out="Y3"/>

<apply-external type="xslt" id="FormatXML" file="biblio/
                                   xmlverbatimwrapper.xsl"
   in="D3" out="Z3"/>
```

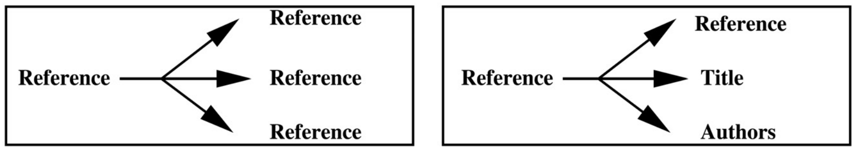

Figure 3 Complex dispatcher

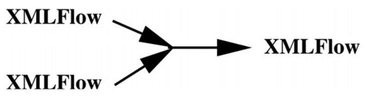

Figure 4 Basic merger

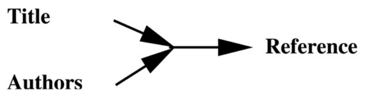

Figure 5 complex merger

3.5 Iterator

Iterators enables the direct expression of the repeated application of the same process. It specifies the control flow so that the output of the previous instance of the process can be the input of the next one. The corresponding construct is `<repeat times='n'> process </repeat>` which repeats the process *n* times. By default, *n* is * which corresponds to applying the process until the output is equal to the input. This is a closure operation.

The iterator is not an abstract process but rather a constructor of the Transmorpher language. It cannot be refined. The iterator can take several input and output channels and a `channels` attribute which allows users to specify the data flow within the iterator: output is copied to the `channels` channels so that it can be compared with the next output.

The bibliography example does not use iterators, so here is a simple example.

```
<process name='myotherprocess' in='channel1,channel2'
                                              out='channel3'>
  <repeat in='channel1' channels='channel4' out='channel3'>
    <apply-process id='p1' name='myprocess' in='channel4'
                                              out='channel4'>
  </repeat>
</process>
```

The classic iterator is implemented by launching the corresponding number of processes. The closure iterator is not implemented yet.

We are currently re-designing the iterators by dividing into the repeat until a condition is satisfied (like our `repeat`) and a proper iterator which iterates an element over a particular range (e.g. number, directory, elements in an XML document...).

4 RULESET

The lower level is made of rules (corresponding to XSLT templates). Instead of using only one kind of very general template, we provide a collection of very simple-to-use and easy-to-analyze rules.

The advantages of rules are that:

- they can be easily understood,
- they can be easily implemented through mapping to XSLT (also efficiently implemented),
- they can be easily analyzed by an external program.

4.1 Rules

The rules themselves are very simple templates that can be used for specifying simple transformations of a source. Here is the current set of rules:

`<maptag match='tag1' target='tag2' />` maps a particular tag (*tag1*) to another one (*tag2*). It can contain remtag, maptag, mapatt, rematt or addatt tags for modifying the content of the matched tags. This is useful for straightforward DTD translation.

```
<maptag match='reference' target='bibitem'
                                  context='bibliography'/>
```

transforms all the reference elements in the context of a `bibliography` element in a `bibitem` element.

`<remtag match='tag' />` simply removes the subtrees rooted in the specified *tag*. This is useful for simplifying the structure of the document.

```
<remtag match="abstract" context="reference"/>
```

suppresses all abstract elements (and all their content) in the context of a `reference` element.

`<flatten match='tag' />` for replacing a tree by its subtrees (or text) in a structure (this rearrangement is useful for information gathered from multiple sources).

```
<flatten match='bibliography'/>
```

suppresses all the bibliography elements within another bibliography element (but not their content).

<mapatt match='name1' target='name2'/> maps a particular attribute to another one.

```
<mapatt match='issue' target='number'
                                        context='conference'/>
```

transforms each issue attribute in the context of a conference element in a number attribute.

<rematt name='name' /> simply remove the attributes specified by *name*.

```
<rematt match="status"/>
<rematt match="isbn" context="book"/>
```

removes all status attributes and all ISBN attributes in the context of a book element.

<addatt name='name' value='value' /> adds to the considered elements a particular attribute and value. This enables operations like decoration of a tree with specific primitives. This is useful for decorating a subtree with a particular attribute (e.g. to color all level 1 titles in red).

```
<addatt name="color" value="red"/>
```

<resubst match='path' source='re' target='ss' /> substitutes each occurrence of the regular expression *re* in the content of the context *path* (usually the value of an attribute or the non structured content of an element) by the substitution string *ss* (which can refer to extracted fragments).

```
<resubst match="conference/@issue" source="([0-9]+)"
                                              target="$1e"/>
```

substitutes each number by the same number followed by "e" in the context of the issue attribute of a conference element. resubst is not pure XSLT. It is implemented as an XSLT extension function and uses the gnu.regexp package.

All these rule tags can use the context attribute, which enables the restriction of the evaluation context of a rule with an XPath location.

4.2 Rulesets

The rule constructs are grouped into rulesets (corresponding to XSLT stylesheets and which can contain regular XSLT templates). The goal of these rulesets is the same as XSLT stylesheets with a restricted set of actions.

In the transformation flow initially provided, there is the strip-abstract ruleset suppressing the abstract elements (and other minor elements) and the elements marked as private.

```
<ruleset name="stripAbstract">
  <remtag match="abstract" context="reference"/>
  <remtag match="keywords" context="reference"/>
  <remtag match="areas" context="reference"/>
  <remtag match="softwares" context="reference"/>
  <remtag match="contracts" context="reference"/>
  <remtag match="*[@status='hidden']"/>
  <resubst match="conference/@issue" source="([0-9]+)"
      target="$1e"/>
  <rematt match="status"/>
  <rematt match="isbn" context="book"/>
</ruleset>
```

The rulesets are transformations that can be invoked in processes. They have one implicit input and one implicit output. Both channels are implicit in the writing of the rules but must named when calling a rule set:

```
<apply-ruleset ref="stripAbstract" id="StripAbstract" in="D1"
    out="X1"/>
```

The ruleset control scheme is exactly the same as that of XSLT: one-pass top-down evaluation. The apply-ruleset tag has a strategy attribute in which we plan to specify other evaluation strategies.

The ruleset implementation consists of transforming the ruleset in a stylesheet that is processed by Xalan. It is easy to see how these rulesets can be transformed into a proper XSLT stylesheet.

The XSLT stylesheet corresponding to the ruleset above is given below:

```
<?xml version="1.0"?>
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
 xmlns:regexp="xalan://
fr.fluxmedia.transmorpher.regexp.RegularExpression">

<!-- **************************************************** -->
<!--    This style sheet was generated by Transmorpher    -->
<!-- **************************************************** -->
  <!-- Copying the root and its attributes -->
  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="@*">
    <xsl:attribute name="{name()}"><xsl:value-of select="."/>
    </xsl:attribute>
  </xsl:template>
  <!-- Copying all elements and attributes -->
  <xsl:template match="*">
    <xsl:copy>
      <xsl:apply-templates select="*|@*|text()"/>
    </xsl:copy>
  </xsl:template>

<!-- ****************************************************** -->
<!-- End of the general section, here begins the stylesheet-->
<!-- ****************************************************** -->

  <!-- Removing elements abstract -->
  <xsl:template match="reference/abstract"/>

  <!-- Removing elements keywords -->
  <xsl:template match="reference/keywords"/>

  <!-- Removing elements areas -->
  <xsl:template match="reference/areas"/>

  <!-- Removing elements softwares -->
  <xsl:template match="reference/softwares"/>

  <!-- Removing elements contracts -->
  <xsl:template match="reference/contracts"/>

  <!-- Removing elements *[@status='hidden'] -->
  <xsl:template match="*[@status='hidden']"/>

  <!-- Substituting all ([0-9]+) by $1e in conference/@issue --
                                                            >
  <xsl:template match ="conference/@issue">
    <xsl:if
         test="function-available('regexp:substitute') and
         function-available('regexp:substituteAll') ">
      <xsl:attribute name="issue">
        <xsl:value-of
           select="regexp:substituteAll(.,'([0-9]+)','$1e')"/>
      </xsl:attribute>
    </xsl:if>
  </xsl:template>

  <!-- Removing attributes status -->
  <xsl:template match="@status"/>

</xsl:stylesheet>
```

5 IMPLEMENTATION

A straightforward implementation consists in transforming every process in a XSLT stylesheet and generating a script for the processing of these stylesheets. The problem is the portability of the scripts: they will involve command lines and tests that are not portable at all.

The system has thus been implemented as a transformation flow interpreter which processes the transformation flow description. It parses the transformation flow description and generates threads for the required processes (starting from `main`) and SAX streams for the channels. It then reads the input files and lets the system process them.

We currently take advantage of Java 1.3, SAX 2.0, Xerces 1.3, Xalan 2 and gnu.regexp 1.1.2. Many transformations are translated into XSLT and passed to Xalan.

5.1 State of the system

The current prototype demonstrates the possibility of implementing the Transmorpher model. It takes advantage of threads and SAX2 event streams. The prototype implements the basic processing model and instances for each kind of process. More precisely, it implements:

- The set of abstract transformations as presented before (generators, serializers, rule set processors, dispatchers, mergers, query evaluators, external processing calls and iterators) and one instance of each interface;
- The rule sets as presented before (modification of tags and attributes, regular expression substitution of content);
- The parsing of transformations flow (in XML) and the creation of the corresponding instances;
- The processing of these systems.

There remain, however, many enhancements to be developed:

- An unbounded evaluator for computing closures;
- Genericity of the Transmorpher itself (using JAXP [Armstrong 2001a]) so that it can use other parsers or XSLT processors than Xerces/Xalan; more generally full compliance with JAXP ("Java API for XML Processing", ex-TRAX) will permit to see a transmorpher process as just another transformation;
- Full UNICODE compatibility testing;
- Servlet interface;
- Cache and cache consistency;
- Dealing with all SAX2 handlers (DTDHandler, ErrorHandler and EntityResolver);
- Static channel type checking. Debugging/stepping mode for Transmorpher processes.

5.2 Performance issues

The goal of Transmorpher was not performance improvement, but better intelligibility. However, so well thought-out technology was available at the time of implementing that we tried to make the most out of it. The main characteristics of this implementation are the use of threads and of SAX2 event streams.

To evaluate Transmorpher performances, we have experimented Transmorpher on a set of XML files representing bibliographical information. The size of XML files ranges from 1kbyte to 1Mbyte. The tests have been run 10 times with the average value taken (figures were very regular). The figures come from a Pentium II/400 running Windows 95.

Four programs have been compared:

Thread Transmorpher, with a version which takes advantage of common computation (for instance, to compute the first and the second output we need to strip the hidden part of the bibliography, this computation is done only once). This version uses threads.

NoThread Transmorpher, with a version which takes advantage of common computation. This version does not use threads.

NoCommonPart Transmorpher, with a version which does not take advantage of common computation. This version uses threads.

Xalan (Java) pipeline The basic pipeline program of Xalan.

The results are described in the two following graphs, on the horizontal axis the XML file sizes, on the vertical axis, the time to generate the four output files.

One can observe a nearly constant gap between the Xalan pipeline and Transmorpher transformation flow. It is related to the number of times Xalan is launched in the Xalan experiment. The increasing difference between the NoThread and NoCommonPart versions can be thought to be related to the increasing importance of the common part while the also increasing difference between Thread and NoThread is related to avoiding numerous disk accesses by the threads.

The second part of the experiments has been performed on moderately small files. The full version of Transmorpher (Thread) is not competitive enough with the other because the time taken by the creation of all the threads is too important with regard to the processing time.

Performances could certainly be improved by using Sun's XSLTC compiler [Jørgensen 2001a]. This compiler is now built-in Xalan 2.1 and not doubt that its full integration within Xalan could be used to offer compiled Transmorpher transformations.

5.3 Availability

Transmorpher is planned to be released very soon under the GNU General Public License.

It can be retrieved (together with its documentation) at http://transmorpher.inrialpes.fr.

6 RELATED WORK

The problems that have been addressed here are generally acknowledged in various contexts and several systems can be compared with Transmorpher:

Perl "practical extraction and report language" (http://www.perl.com) is typically a competition for XSLT on the non structured content of documents. Several possible tasks in rulesets are inspired from Perl (and especially regular expression substitutions). Moreover, Perl allows the description of subroutines and the composition of these subroutines (in a procedural or functional manner) [Wall 2000a]. Perl retains two main problems: it is not far away from a full-fledged programming language necessitating training of the users and it is not very strong on structured transformations.

Bladerunner (http://www.xmlecontent.com) was an XML flow management system integrating database access and allowing XSLT transformations. It generates output in PDF, HTML, PostScript. Its main weakness was to be a proprietary system used for consulting activities. Its specifications are not far away from those of Transmorpher but information

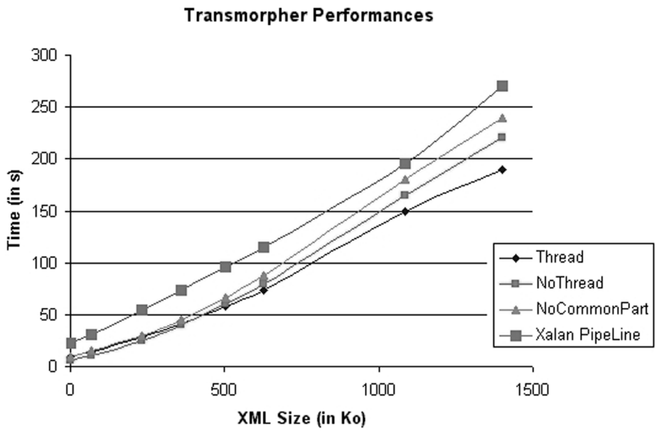

Figure 6 Transmorpher performances (overall)

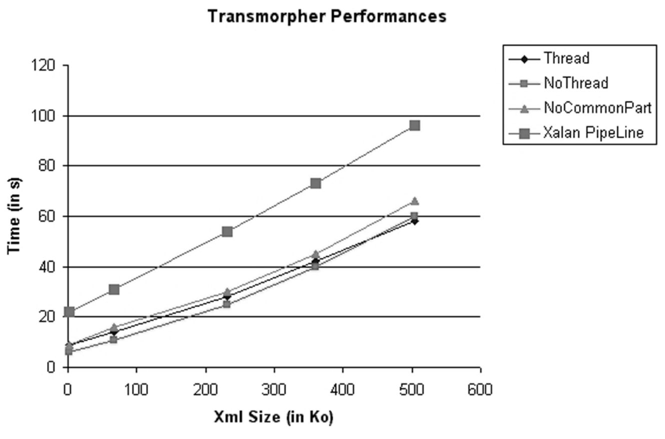

Figure 7 Transmorpher performances (small files)

on how it works is not readily available (it has been suppressed from the product list of Broadvision).

Only a few systems enables the organization of several transformations at once. One can cite AxKit (http://www.axkit.com, [Sergeant 2000a] [Sergeant 2000b]) an extension of Apache written in Perl which can determine the stylesheet to use in order to generate a requested document and can pipeline several transformations. A plug-in architecture allows the use of processors other than XSLT. However, there seems to be no explicit description of the transformation flow (which is deduced from processing-instructions within the documents). It is thus difficult to analyze transformations (in order to optimize, compile or verify them).

W3Make (http://www.skamphausen.de/software/w3make/) is aimed at using Makefiles for statically updating web sites. It enables the definition of generic processes with the help of computation templates and it is able to compute the minimal part of the flow to be recomputed. W3Make is file oriented and thus specifies how to compute a particular file (rather than a process).

XotW [Ståldal 2000a] is aimed at generating web sites offline. The site is described by a sitemap providing all the directories and files in the web site. XotW distinguishes between five types of components: format (a serializer), split (a dispatcher), transform (a transformation), source (a generator) and copy (which does not exist in Transmorpher and is related to the fact that XotW deals with files). XotW has Make-like facilities for recomputing files only when necessary. Emphasis is put on this evolved for of caching. Like Make, XotW mixes sitemap and stylesheet for all the files. The transformations are rules for providing a file. They are expressed in a functional way (each producer, but split, can only provide one file). This prohibits transformation flow reuse.

Cocoon (http://xml.apache.org/cocoon, [McLaughlin 2000a]) is another stylesheet composition system written in Java and integrated in any Servlet server. Advantages of Cocoon include document caching and explicit declaration of transformations ("sitemap"). Cocoon is based on a three-step site publication model (creation, content processing and rendering). This provides a clear methodology for developing sites but confines the system to a particular type of processing. The caching mechanism of Cocoon is tied to that methodology by enabling caching only at these steps.

Cocoon, XotW and AxKit are relatively dedicated to web site development. Consequently they adhere to the demand-driven (pull) model and optimize pipelining (instead of more complex transformation flows). They develop elaborate caching policies which are coherent with their applications. Such caching possibilities could be implemented in Transmorpher.

Concerning the comparison with particular transformation schemes (i.e. rulesets), there has been much work in term rewriting that can be applied to XML transformations. Stratego (http://www.stratego-language.org, [Visser 2000a]) and Elan (http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/, [Borovansky 2001a]) are dedicated to the expression of strategy in rewrite rule processing. They consider assembling small rules through traversal strategies instead of applying a monolithic stylesheet. This kind of strategies could be adapted to the ruleset processing (and also to XSLT processing) with profit.

Concerning work on XSLT simplification, Paul Tchistopolskii (http://www.pault.com, [Tchistopolskii 2000b]) developed XSLScripts. The XSLS (a.k.a. XSLT Scripts) is a non-XML based language that can write any XSLT stylesheet. It is a more concise language than XSLT but still a complex one.

7 CONCLUSION

We have presented the Transmorpher system and principles which aim at complementing XSLT on the issue of simple transformations and complex control of transformations. Although only limited experiments have been conducted with the system it behaves the way it was intended. The concepts that have been proposed in this paper are viable ones and overcomes some limitations of XSLT (more generally, of stand-alone one-pass template-only transformation systems).

Like Cocoon demonstrates, Transmorpher corresponds to a real need and we expect to see more systems of its kind in the next few years. The bibliography example is a real example available from our web site.

It is of notice, that Transmorpher has not been designed for efficiency but for intelligibility: Our ultimate goal is to offer an environment in which we will be able to certify that a particular output of a compound transformation satisfies some property (e.g. hiding some type of information, preserving order, or preserving "meaning"). Another good property of intelligibility (through modularity) is that it is easy to imagine a graphical user interface to Transmorpher.

Many possible improvements have been proposed above and will help Transmorpher evolve.

This work has been strongly supported by FluxMedia.

BIBLIOGRAPHY

[**Jørgensen 2001a**] Morten Jørgensen, The XSLT compiler for the JVM, Proc. 1st XSLT-UK conference, Oxford (UK), 2001.

[**Ståldal 2000a**] Mikael Ståldal, Presenting XML documents on different media with stylesheets, Master's thesis, KTH, Stockholm (SE), 2000.

[**Armstrong 2001a**] Eric Armstrong, Working with XML: the Java API for XML Parsing (JAXP) Tutorial, 2001, http://java.sun.com/xml/jaxp-docs-1.0.1/docs/tutorial/

[**Borovansky 2001a**] Peter Borovansky, Claude Kirchner, Hélène Kirchner and Christophe Ringeissen, Rewriting with strategies in ELAN: a functional semantics, *International Journal of Foundations of Computer Science* 12(1):69–96, 2001.

[**Clark 1999a**] James Clark (ed.), XSL transformations (XSLT) version 1.0, W3C Recommendation, 1999, http://www.w3.org/TR/xslt

[**Clark 1999b**] James Clark, Steve DeRose (ed.), XML Path Language (XPath) version 1.0, W3C Recommendation, 2001, http://www.w3.org/TR/xpath.html

[**Clark 2001a**] James Clark (ed.), XSL transformations (XSLT) version 1.1, W3C working draft, 2001, http://www.w3.org/TR/xslt11/

[**Kay 2000a**] Michael Kay, XSLT Programmer's reference, Wrox press, Birmimgham (UK), 2000.

[**Becker 2000a**] Oliver Becker, XSLT, 2000, http://www.informatik.hu-berlin.de/~obecker/XSLT/

[**Drewes 2000a**] Frank Drewes and Peter Knirsch and Hans-Jörg Kreowski and Sabine Kuske, Graph transformation modules and their composition, Proceedings of international workshop ACTIVE 99 (*Lecture notes in computer science* 1779), pp. 182–191, 2000.

[**Boag 2000a**] Scott Boag, TRaX and Serialize APIs, 2000, http://trax.openxml.org

[**Wall 2000a**] Larry Wall, Tom Christiansen, Jon Orwant, Programming Perl, 3rd Edition, O'Reilly and associates, Sebastopol (CA US), 2000.

[**Sergeant 2000a**] Matt Sergeant, How AxKit works, O'Reilly and associates, 2000, http://www.xml.com/pub/2000/05/24/axkit/index.html

[**Sergeant 2000b**] Matt Sergeant, AxKit: XML web publishing with Apache and mod_perl, O'Reilly and associates, 2000, http://www.xml.com/pub/2000/05/24/axkit/index2.html

[**McLaughlin 2000a**] Brett McLaughlin, Web Publishing Frameworks, in: Brett McLaughlin, Java and XML, O'Reilly and associates, Sebastopol (CA US), 2000, http://www.oreilly.com/catalog/javaxml/chapter/ch09.html

[**Visser 2000a**] Eelco Visser, Strategic Pattern Matching, Proceedings of Rewriting Techniques and Applications, pp.30–44, Trento (IT),(*Lecture notes in computer science* 1631), 1999.

[**Tchistopolskii 2000b**] Paul Tchistopoloskii, XSLScripts, 2000, http://www.pault.com/xsls