Article

XML transformation flow processing

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

Laurent Tardif
Monash University
Clayton, Australia
WEB http://www.csse.monash.edu.au/~tardif/
EMAIL laurent.tardif@csse.monash.edu.au

XML has stimulated the publication of open and structured data. Because these documents are exchanged across software and organizations, they need to be transformed in many ways. For that purpose, XSLT allows the development of complex transformations that are adapted to the XML structure. However, 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.

Introduction and motivation

In electronic documentation, a transformation is a process that transforms a source document into another document, the target. The notion of a transformation has been used for some time [Furuta/Stotts 1988], [ISO 1991] and now 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 present a system that targets increased intelligibility in the expression of transformations. We give the motivation for this system 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.

Jérôme Euzenat and Laurent Tardif, “XML transformation flow processing”
Markup Languages: Theory & Practice 3.3 (2001): 285–311

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 years 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 first two documents must initially have 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, elements marked as private, and 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.

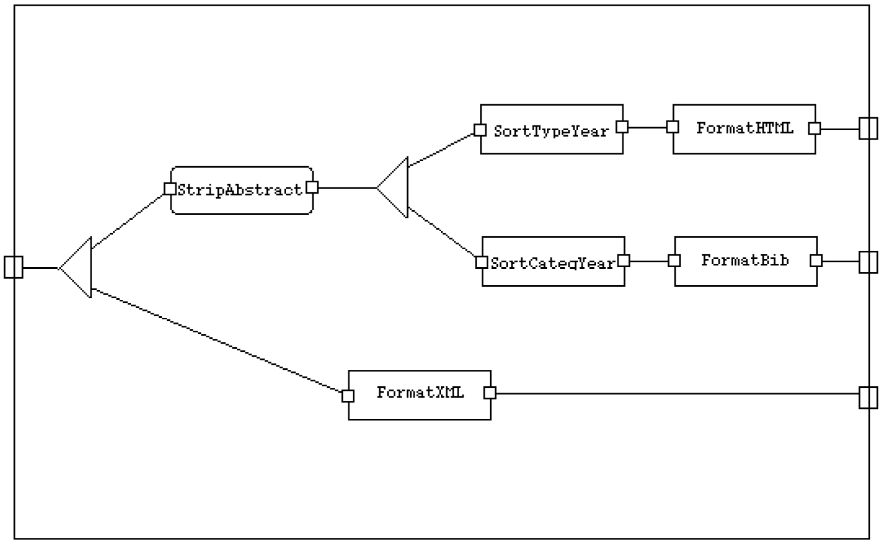

Figure 1 | A sample transformation flow (flowing from left to right)

Limitations of XSLT

XSLT [W3C 1999b] 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:

Complexity: Writing simple transformations (*e.g.*, tag translation, tree decoration, or 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 [W3C 1999a], XSLT [W3C 1999b] adds seven extension functions, including `document()` for processing multiple documents. The treatment of multi-source transformations, through this `document()` construct, contributes to XSLT's lack of intelligibility. Moreover, there is a strong inequality of treatment vis-a-vis output specification that is dealt with through an explicit `output` XSLT construct. (Although this could change a bit with further revision of the XSLT Recommendation [W3C 2001], this construct is not in a totally satisfying.)

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.

No self-sufficiency: Writing complex transformation flows involving multiple documents, independently designed stylesheets, and closure operations requires the use of an external environment (*i.e.*, a 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 as 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 [Kay 2000], [Becker 2000].

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.

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, . . .);
- allow regular expression transformations on the content;
- compose transformations by linking their (multiple) output to input;
- iterate transformations, sometimes until saturation (closure operation); and
- integrate external transformations.

The guidelines of the proposal are the following:

- be as compatible as possible with XSLT (by defining transformations from many Transmorpher elements to XSLT);
- be portable, *i.e.*, implementation in Java (as opposed to generating an XSLT stylesheet(s) and scripts which would have been an easy solution);
- be open to other systems by importing any kind of stylesheet and expressing control over it; and
- have 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.

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 information in the form of SAX events [Boag 2000]). They can take several inputs and provide several outputs during one execution. The Transmorpher computing model is thus rather simple. In the following, each component of the computing model is briefly described.

Processes (or transformation flows)

The transformation flows can be specified by programming the class instantiation in Java or by describing it in XML. Below is an XML transformation flow, in which the `processGeneral` process corresponds to the flow described by Figure 1 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"
                   in="X11" out="X2">
     <with-param name="file">biblio/sort-ty.xsl</with-param>
   </apply-external>

   <apply-external type="xslt" id="FormatHTML" in="X2"
                   out="X3">
     <with-param name="file">biblio/form-harea.xsl</with-param>
   </apply-external>

   <apply-external type="xslt" id="SortCategYear"
                   in="X12" out="Y2">
     <with-param name="file">biblio/sort-cya.xsl</with-param>
   </apply-external>

   <apply-external type="xslt" id="FormatBib" in="Y2" out="Y3">
     <with-param name="file">biblio/form-bibtex.xsl</with-param>
   </apply-external>

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


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

<main name="ProcessBiblio">
  <param name="name">biblio</param>
  <generate type="readfile" id="bibexmo" out="R1">
     <with-param name="file">biblio/biblio.xml</with-param>
  </generate>
  <generate type="readfile" id="je" out="R2">
```

```
       <with-param name="file">biblio/je.xml</with-param>
    </generate>
    <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">
       <with-param name="file">${name}.html</with-param>
    </serialize>
    <serialize type="writefile" id="writeBIB" in="Y31">
       <with-param name="file">${name}.bib"</with-param>
    </serialize>
    <serialize type="writefile" id="writeXML" in="Z31">
       <with-param name="file">${name}-xml.html</with-param>
    </serialize>

    <apply-process id="processTB" ref="processByNames"
                   in="X12" out="W3" />
    <serialize type="writefile" id="writeTB" in="W3">
       <with-param name="file">${name}_tb.html</with-param>
    </serialize>
  </main>

</transmorpher>
```

The transformation flows are described through a *process* element. There can be several such processes in one document. Other kinds of dedicated Transformation flows appearing in a Transmorpher document are *main*, *servlet*, and *transformer*. They indicate how to compile the document.

The processes specify a transformation flow by defining a set of transformations to be processed, the flow of information from transformation to transformation, and the variables (parameters) that are available in the process. The processes can have input or output channels, but do not define a control structure: the flow of information implicitly specifies the valid execution orders for the transformations.

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, integrated in another application, or used 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.

Transformations

The processes contain a set of subprocesses or transformations. These transformations are identified by an element corresponding to its computing model. Currently, the available transformations are: generators, serializers, rule set processors, dispatchers, mergers, query evaluators, external processing calls, and iterators. Each of these primitives has an `id` (enabling the identification of subprocesses of the same kind) and usually *`in`* and *`out`* attributes (enabling their connection to other processes). The *`ref`* attribute denotes an element defined within the current transformation (or an imported one). The *`type`* attribute identifies a particular implementation of the basic process.

The most important types of subprocesses are:

- `<apply-process ref='name'/>` which calls an already defined process,
- `<apply-ruleset ref='name' strategy='strategy' />` which applies a set of rules (equivalent to an XSLT stylesheet) to its input,
- `<apply-external type='type' />` which calls an external procedure on the input and must provide the output (This engine could be Perl, XSLT, or whatever is appropriate.), and
- `<apply-query ref='name' type='type' />` which evaluates a query on the input and must provide the output (This query engine could be XQL, SQL, or whatever is appropriate.).

These transformations refer to entities that have been defined by the user (other processes or XSLT stylesheets).

Transmorpher also provides a set of abstract elementary transformations that are provided with an interface and execution model. They can be called by the user through the corresponding elements:

- `<dispatch type='type'/>` which takes one input and several outputs,
- `<merge type='type'/>` which takes several inputs and one output,
- `<generate type='type'/>` which takes no input and one output (generally used to read from outer streams like files), and
- `<serialize type='type'/>` which takes one input and no output (generally used to write to outer streams like files).

The end user cannot introduce new elementary transformations (contrary to processes and rulesets that are described by the user). The user can only use the implementations of these transformation that are available. A developer can add a new implementation of such a transformation by creating a new class corresponding to its interface. This new implementation will be available by changing the *`type`* attribute.

Channels

In Transmorpher the generic processes can have several input and output ports. These ports are connected to channels that are fed by the output of a process and can be used as input of other processes. They are abstractions which enable 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 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 Frank Drewes and his coauthors [Drewes et al. 2000] considers 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, rather than 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. Thus, they 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.

Variables

Variables are used for parameterizing the behavior of Transmorpher. They follow the parameter syntax of XSLT. Variables take string values which are passed from process to process through the call of a process.

The variables are declared in the definition of a process with a `param` element which can declare default values for them. They are explicitly bound during the call of a transformation by the `with-param` element.

Custom transformation implementation (*e.g.*, a new implementation of a `serialize` element) can freely add new variables to those defined by Transmorpher (`e.g.`, an encoding parameter). The Transmorpher DTD does not have to be changed for this new variable, and Transmorpher processing will have the same behavior.

Variables are evaluated at run time. This provide more flexibility in parameterization. For that purpose, variables should be communicated to the running program. This can be done for compiled Java runtime by

```
java fr.fluxmedia.transmorpher.Application.transmorph \\
  -Dname=biblio process.xml
```

for servlet by

```
http://ServletServerURL/ProcessBiblio?name=biblio
```

and for JAXP Transformer by the Java snippet

```
Transformer transmorpher = new ProcessBiblio();
transmorpher.setParameter( "name", "biblio" );
transmorpher.transform( in, out );
```

As far as Transmorpher is concerned, variables are only passed from one process to another and expanded in other variable values. The true consumers of the variables are the process implementations (like the *readfile* dispatcher or the *xslt* external process which can get their variables from Transmorpher variables).

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 and 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.

Generator and serializers

When a transformation flow takes input from a source which is not a stream (*e.g.*, 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 (or 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">
       <with-param name="file">biblio/biblio.xml</with-param>
    </generate>
```

```
<generate type="readfile" id="je" out="R2">
   <with-param name="file">biblio/je.xml</with-param>
</generate>
```

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

```
<serialize type="writefile" id="writeHTML" in="X31">
   <with-param name="file">${name}.html</with-param>
</serialize>
<serialize type="writefile" id="writeBIB" in="Y31">
   <with-param name="file">${name}.bib"</with-param>
</serialize>
<serialize type="writefile" id="writeXML" in="Z31">
   <with-param name="file">${name}-xml.html</with-param>
</serialize>
<serialize type="writefile" id="writeTB" in="W3">
   <with-param name="file">${name}_tb.html</with-param>
</serialize>
```

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

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.

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 a list of authors. This can be useful for generating a list of authors (sorted and "uniqued") 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.

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.

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 by taking 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

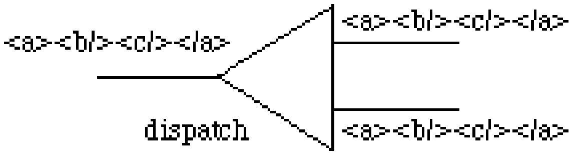

Figure 2 | The built-in *broadcast* dispatcher duplicates the input to the output channels.

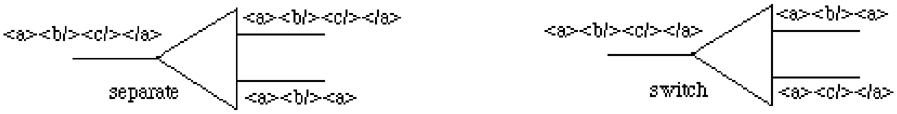

Figure 3 | Two custom dispatchers. The *switch*separates some elements from the others; this can be used for separating a bibliography with regard to the category of publications. The *separator* duplicates the input on the first output and sends specific elements on the others; this can be used for sending a book on the first channel, and its chapter and section titles on the other for building the table of contents.

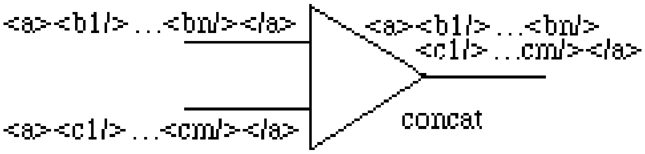

Figure 4 | The built-in merger *concat* simply concatenates twoXML files under the root of the first file. It can be used for merging the publication lists of several researchers in a group.

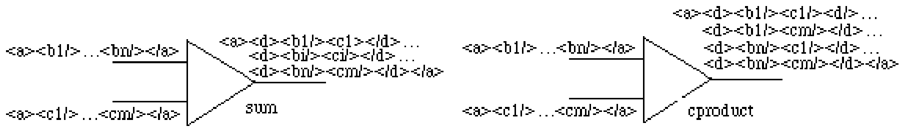

Figure 5 | Two examples of custom mergers. The *sum* can be used for building references out of a list of authors and a list of papers they wrote. The cartesian product can be used for generating all the combinations of desktop computers from a list of **CPU** boards and a list of

select elements. The *select* elements have a *match* attribute containing an XPath [W3C 1999a] 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 nor 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>
```

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

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 additional parameters.

Transmorpher provides an XSLT external process type which enables the processing of XSLT stylesheets already defined. (This was the case of the stylesheets in 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" in="X11" out="X2">
  <with-param name="file">biblio/sort-ty.xsl</with-param>
</apply-external>
<apply-external type="xslt" id="FormatHTML" in="X2" out="X3">
  <with-param name="file">biblio/form-harea.xsl</with-param>
</apply-external>

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

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

Iterator

Iterators enable the direct expression of a repeated application of the same process. They specify 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> iterator* process </repeat>` which iterates variables over the structure defined in an iterator and executes the process at each iteration. The iterators are specified in the *`iterator`* construct which introduces a variable and defines the trajectory of the variable.

The iterator can take several input and output channels but have some specific kinds of channels: the feedback channels which denote that the content output to the channel at iteration i will be the input of the channel at iteration $i+1$.

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

```
<repeat id="rep" in="i1 i2" out="o1 o2" bin="b1 b2" bout="b2 b1">
  <iterate type="int" name="i">
    <with-param name="from">0</with-param>
    <with-param name="to">5</with-param>
  </iterate>
  <iterate type="file" name="f">
    <with-param name="indir">../input</with-param>
    <with-param name="filter">*.xml</with-param>
  </iterate>
  <apply-process id='p1' ref='myprocess' in='b1' out='b2'>
    <with-param name="filename" select="${f}${i}"/with-param>
  </apply-process>
  <apply-process id='p2' ref='myprocess' in='b2' out='b1'>
    <with-param name="filename" select="${f}${i}"/with-param>
  </apply-process>
</repeat>
```

The example iterates the process at most 5 times on the XML file names present in the input directory. It copies its first input in feedback channels *b1* and *b2*. After one iteration, these channels are feed with output of the *p1* and *p2* processes.

The *repeat* construct is not an abstract process but rather a constructor of the Transmorpher language. It cannot be refined. The iterators are defined by the element *iterate*. They have a *type* attribute defining the type of the variable and the way it iterates, and a *name* attribute for specifying the name of the variable. This variable can be used like other variables in the body of the transformation. Currently, we have only implemented an iterator on integer values, but the iterators should be refinable by a developer. The parameters of the iterator are then passed to it as parameters.

The iterators iterate on all the variables introduced and stop as soon as a variable has finished iterating. If there is no iterator, then Transmorpher applies the process until some output (defined in a *test* attribute) is equal to its previous value in the iteration. This is a closure operation.

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-analyse rules.

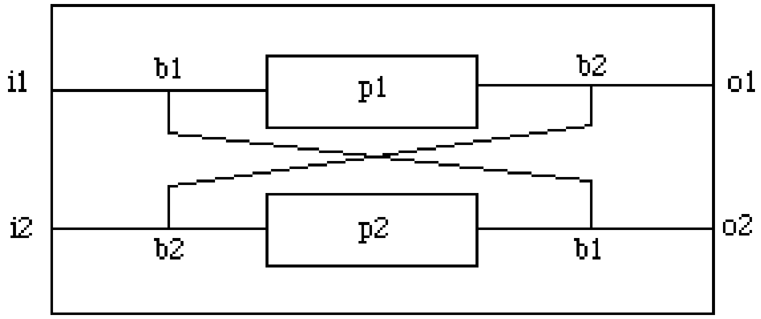

Figure 6 | The depiction of the described iterator

The advantages of rules are:

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

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. The string

```
<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. The string

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

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

`<flatten match='tag' />`*:* Replaces a tree with its subtrees (or text) in a structure. (This rearrangement is useful for information gathered from multiple sources.) The string

```
<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. The string

```
<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 removes the attributes specified by *name*. The string

```
<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 a particular attribute and value to the elements considered. This enables operations like decoration of a tree with specific primitives. This is useful for decorating a subtree with a particular attribute, for example, 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 nonstructured content of an element) by the substitution string *ss* (which can refer to extracted fragments). The string

```
<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. The 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.

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 be 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>
```

Implementation

A straightforward implementation of the Transmorpher model consists of transforming every process in an 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.

The system is made of two parts:

- An internal representation of the transformation as specified in XML. This representatation can be manipulated by a public API (enabling the future implementation of a graphical user interface). It can load and save a transformation description in XML, and build the classes required for running the transformation or generating a Java program for executing this transformation.
- A runtime library (execution classes) that contains the runtime implementation of the built-in transformations. This library is independent from the internal representation. It can be extended without modifying the internal representation.

We currently take advantage of Java 1.3, SAX 2.0, Xerces 1.3, Xalan 2.2, and gnu.regexp 1.1.3. Many transformations are translated into XSLT and passed to Xalan.

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; and

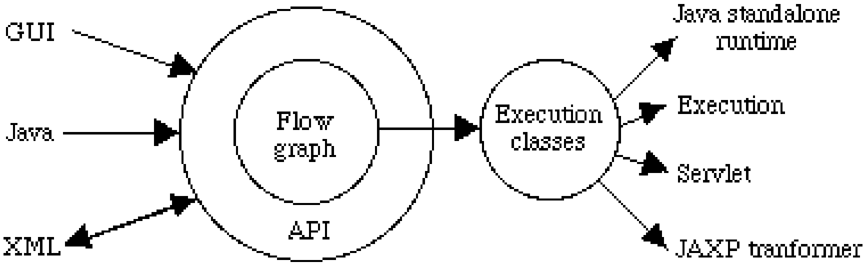

Figure 7 | The architecture of our implementation of Transmorpher revolves around a graph representing the transformation flows that can be manipulated by an API. This graph can be built with the API, by parsing an XMLdescription of the flow or through a GUI (to be implemented). It allows the generation of an XML representation of itself or Java code for either immediate execution or compilation as a servlet, a JAXP transformer, or a standalone transformation.

- the processing of these systems or their “compilation” to a Java standalone source, a JAXP transformer class [Armstrong 2001], or a servlet class.

Iterators are not fully tested, and the unbounded evaluator for computing closures is not yet implemented. All SAX2 handlers (*DTDHandler*, *ErrorHandler* and *EntityResolver*) are not yet available all the way. Some tests like full UNICODE compatibility must also be carried out.

There remain many possible enhancements that could be investigated in the Transmorpher framework:

- XSLT compilation (XSLTc), cache, and cache consistency;
- Rule evaluation strategies;
- DOM-based computation, key propagation, and incremental transformations (continuous diff-based flows) [Villard/Layaïda 2001];
- Full UNICODE compatibility testing; and
- Static channel type checking; debugging/stepping mode for Transmorpher processes.

Performance issues

The goal of Transmorpher was not performance improvement, but better intelligibility. However, such 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 SAX2 event streams.

To evaluate Transmorpher performance, we have tested Transmorpher on a set of XML files representing bibliographical information passed through the transformation given in the example. The size of the XML files ranges from 22kbytes to 4.6Mbytes. The tests have been run 10 times with the average value taken (figures were very regular). The figures come from a Pentium IV 1.4GHz running Windows 2000 with JDK 1.3. The compared software is Xalan-Java that is also used in Transmorpher.

Three programs have been compared :

Xalan: The Transmorpher flow is programmed in Java with intermediary results stored in files.

Xalan pipeline: The transmorpher flow is programmed in Java. The results of transformation are pipelined from one Transformer to another. In both Xalan cases, the merge operation is implemented through an additional simple stylesheet.

Transmorpher: Transmorpher with the flow described in XML (It is thus parsed, a graph is generated, and then it is executed.). So far the compiled version is only slightly more rapid. For technical reasons, this test did not use the multithreaded version of Transmorpher (*see* [Euzenat/Tardif 2001]).

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 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. (Certain Transformations are made twice).

The second part of the experiment was performed on moderately small files. On small files, Transmorpher is not competitive enough.

In conclusion, the differences between these systems are not really significant. These tests should be taken as a demonstration that Transmorpher behaves correctly with regard to the systems it uses and did not slow their performances. They also show that avoiding duplication, without caching, is beneficial.

Performances could certainly be improved by using Sun's XSLTC compiler [Jørgensen 2001]. This compiler is now built-in Xalan 2.1 and no doubt its full integration within Xalan could be used to offer fully compiled Transmorpher transformations.

Availability

Transmorpher has been jointly developed by INRIA Rhône-Alpes and Fluxmedia. Transmorpher is released under the GNU General Public License. It can be retrieved (together with its documentation) at <`http://transmorpher.inrialpes.fr`>.

Related work

The problems that have been addressed here are generally acknowledged in various contexts, and several systems can be compared with Transmorpher. In sum-

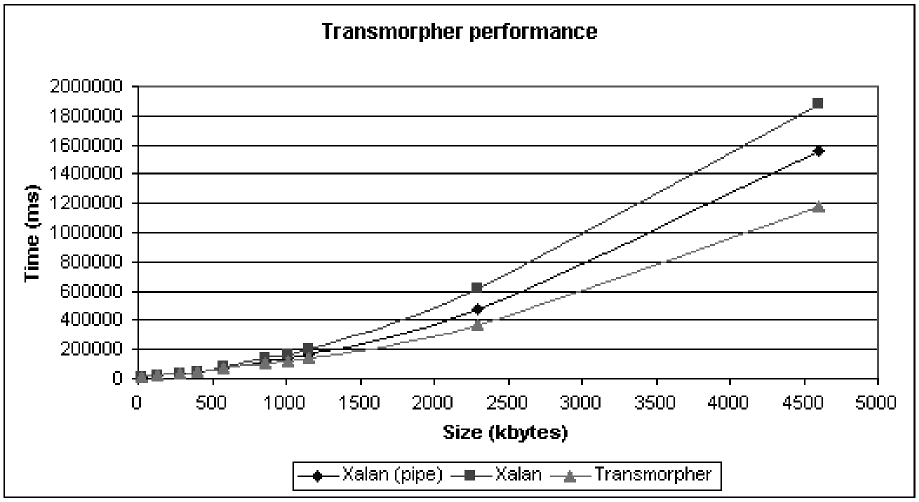

Figure 8 | Transmorpher performances (overall)

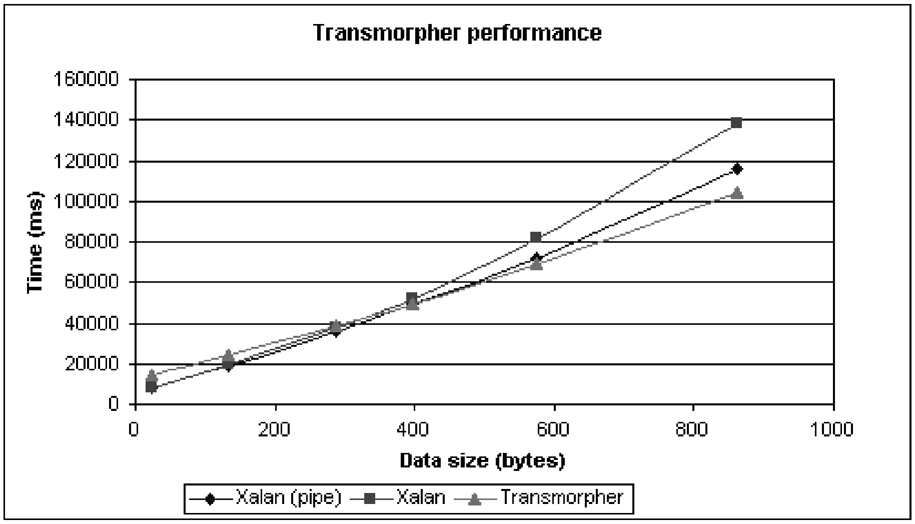

Figure 9 | Transmorpher performances (small files)

mary, the more relevant comparisons are related to XSLT itself and Cocoon, as shown below.

Perl "practical extraction and report language" (<`http://www.perl.com`>) is typically a competitor of XSLT for the nonstructured content of documents. Several possible tasks in rulesets are inspired by 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 et al. 2000]. 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

Interleaf's Bladerunner (<http://www.interleaf.com>) is an XML flow management system integrating database access and allowing XSLT transformations. It generates output in PDF, HTML, and PostScript. Its main weakness is being a proprietary system. It seems similar to Transmorpher, but information on how it works is not readily available. Wizen's PowerXML (<http://www.wizen.com>) offers many details on input and output features, but not on the inside of the system.

Only a few systems enable the organization of several transformations at once. One can cite AxKit (<http://www.axkit.com>; [Sergeant 2000b], [Sergeant 2000a]) 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 analyse transformations (in order to optimize, compile, or verify them).

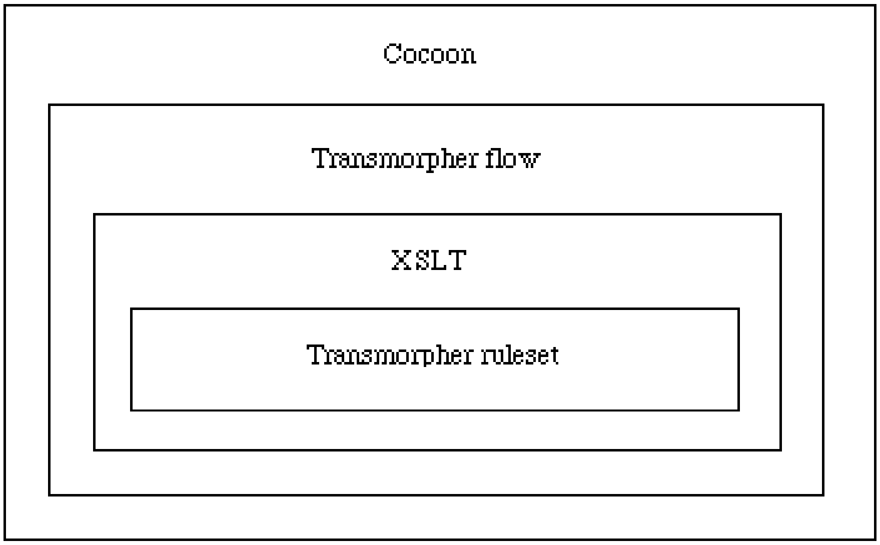

Figure 10 | The relationships between Transmorpher and other tools. The rule part of Transmorpher is a simplification of XSLT, but the flow part is more powerful. It could be used as the Transformation part of Cocoon.

Ant (<`http://jakarta.apache.org/ant/`>) is a substitute for the famous Make using XML and Java. It has several similarities with Transmorpher: a simple processing model and an easily customizable philosophy. Ant implements the *`_null_`* retropropagation from the begining (This is an update tool.). While Transmorpher is flow and data-oriented, Ant is task and file-oriented. Transmorpher is better on data-driven processing while Ant is better in demand-driven.

XotW [Ståldal 2000] 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 form of caching. Like Make, XotW mixes use of a 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 2000]) 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 (a “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. Below is a cocoon sitemap example that achieves the same purpose as a part of our example:

```
<map:process match="htdocs/*.html">
  <map:generate src="{1}.xml" />
  <map:transform src="xslt/strip-abstracts.xsl"/>
  <map:transform src="xslt/sort-year.xsl"/>
  <map:transform src="xslt/form-html.xsl"/>
  <map:serialize type="html"/>
</map:process>
```

As can be seen, even the syntax of both Transmorpher and Cocoon is very close. The basic differences are the absence of input/output specifications which are implicit and the *`match`* attribute for specifying a target.

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. Elan (<http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/>, [Borovansky et al. 2001]) and Stratego (<http://www.stratego-language.org>, [Visser 1999]) 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. These kinds of strategies could be profitably adapted to the ruleset processing (and also to XSLT processing).

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

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. It tries to implement for transformations, the slogan "Make simple things easy and complex ones possible".

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 overcome some limitations of XSLT (more generally, of stand-alone one-pass template-only transformation systems).

As 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.

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.

Acknowledgments

The support of the Fluxmedia Company has been decisive to the development of Transmorpher. It is gratefully acknowledged.

References

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

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

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

[Borovansky et al. 2001] Borovansky, Peter, Claude Kirchner, Hélène Kirchner, and Christophe Ringeissen. 2001. “Rewriting with Strategies in ELAN: a Functional Semantics”. *International Journal of Foundations of Computer Science* 12, no.1:69–96.

[Drewes et al. 2000] Drewes, Frank, Peter Knirsch, Hans-Jörg Kreowski, and Sabine Kuske. 2000. Graph Transformation Modules and Their Composition. In *Proceedings of International Workshop ACTIVE 99*. Also available from *Lecture Notes in Computer Science* 1779:182–191.

[Euzenat/Tardif 2001] Euzenat, Jérôme, and Laurent Tardif. 2001. Processing XML Transformation Flows. In *Proceedings of Extreme Markup Languages 2001*. Arlington, VA: IdeAlliance Association.

[Furuta/Stotts 1988] Furuta, R., and P. Stotts. 1988. Specifying Structured Document Transformations. In *Proc. International Conference on Electronic Publishing* . Nice, France.

[ISO 1991] International Organization for Standardization (ISO). 1991. *ISO/IEC/DIS 10179 Information Technology — Text and Office Systems — Document Style Semantics and Specification Language (DSSSL)*. Genève; New York: International Organization for Standardization.

[Jørgensen 2001] Jørgensen, Morten. 2001. The XSLT Compiler for the JVM. In *Proc. 1st XSLT-UK Conference*. Oxford, United Kingdom.

[Kay 2000] Kay, Michael. 2000. *XSLT Programmer’s Reference*. Birmingham, United Kingdom: Wrox Press.

[McLaughlin 2000] McLaughlin, Brett. 2000. Web Publishing Frameworks. In *Java and XML*. [online]. Sebastopol, CA: O’Reilly and Associates. Available from < http://www.oreilly.com/catalog/javaxml/chapter/ch09.html>.

[Sergeant 2000a] Sergeant, Matt. *AxKit: XML Web Publishing with Apache and mod_perl*. Sebastopol, CA: O’Reilly and Associates. Available from < http://www.xml.com/pub/2000/05/24/axkit/index2.html>.

[Sergeant 2000b] Sergeant, Matt. *How AxKit Works*. Sebastopol, CA: O’Reilly and Associates. Available from < http://www.xml.com/pub/2000/05/24/axkit/index.html>.

[Ståldal 2000] Ståldal, Mikael. “Presenting XML documents on different media with stylesheets”. Master’s Thesis. KTH, Stockholm, Sweden.

[Tchistopolskii 2000] Tchistopoloskii, Paul. 2000. [online]. *XSLScripts*. Available from < http://www.pault.com/xsls>.

[Villard/Layaïda 2001] Villard, Lionel, and Nabil Layaïda. 2001. “iXSLT: an Incremental XSLT Transformation Processor for XML Document Manipulation” (submitted).

[Visser 1999] Visser, Eelco. 1999. Strategic Pattern Matching. In *Proceedings of Rewriting Techniques and Applications*. Trento, Italy. Also available from *Lecture Notes in Computer Science* 1631.

[Wall et al. 2000] Wall, Larry, Tom Christiansen, and Jon Orwant. 2000. *Programming Perl*. 3rd ed. Sebastopol, CA: O’Reilly and Associates.

[W3C 1999a] World Wide Web Consortium (W3C). 1999. *XML Path Language (XPath) Version 1.0*. Edited by James Clark and Steve DeRose. W3C Recommendation 16 November 1999 [online]. User Interface Domain. N.p.: World Wide Web Consortium, 16 November 1999 [cited 2001]. Available from < http://www.w3.org/TR/xpath.html>.

[W3C 1999b] World Wide Web Consortium (W3C). 1999. *XSL Transformations (XSLT) Version 1.0*. Edited by James Clark. W3C Recommendation 1999 [online]. User Interface Domain. N.p.: World Wide Web Consortium, 1999. Available from < http://www.w3.org/TR/xslt>.

[W3C 2001] World Wide Web Consortium (W3C). 2001. *XSL Transformations (XSLT) Version 2.0*. Edited by Michael Kay. W3C Working Draft [online]. User Interface Domain. N.p.: World Wide Web Consortium, 2001. Available from < http://www.w3.org/TR/xslt20/>.