

XMLTrans: a Java-based XML Transformation Language for Structured Data

Derek Walker and Dominique Petitpierre and Susan Armstrong
{Derek.Walker,Dominique.Petitpierre,Susan.Armstrong}@issco.unige.ch
ISSCO, University of Geneva
40 blvd. du Pont d'Arve
CH-1211 Geneva 4
Switzerland

Abstract

The recently completed MLIS DicoPro project addressed the need for a uniform, platform-independent interface for accessing multiple dictionaries and other lexical resources via the Internet/intranets. Lexical data supplied by dictionary publishers for the project was in a variety of SGML formats. In order to transform this data to a convenient standard format (HTML), a high level transformation language was developed. This language is simple to use, yet powerful enough to perform complex transformations not possible with similar transformation tools. XMLTrans provides rooted/recursive transductions, similar to transducers used for natural language translation. The tool is written in standard Java and is available to the general public.

1 Introduction

The MLIS DicoPro project¹, which ran from April 1998 to Sept 1999, addressed the need for a uniform, platform-independent interface for accessing multiple dictionaries and other lexical resources via the Internet/intranets. One project deliverable was a client-server tool enabling translators and other language professionals connected to an intranet to consult dictionaries and related lexical data from multiple sources.

Dictionary data was supplied by participating dictionary publishers in a variety of proprietary formats². One important DicoPro module was a transformation language capable of

standardizing the variety of lexical data. The language needed to be straightforward enough for a non-programmer to master, yet powerful enough to perform all the transformations necessary to achieve the desired output. The result of our efforts, XMLTrans, takes as input a well-formed XML file and a file containing a set of transformation rules and gives as output the application of the rules to the input file. The transducer was designed for the processing of large XML files, keeping only the minimum necessary part of the document in memory at all times. This tool should be of use for anyone wishing to transform large amounts of (particularly lexical) data from one XML representation to another.

At the time XMLTrans was being developed (mid 1998), XML was only an emerging standard. As a consequence, we first looked to more established SGML resources to find a suitable transformation tool. Initial experimentation began with DSSSL (Bingham, 1996) as a possible solution. Some time was invested in developing a user-friendly “front-end” to the DSSSL engine *jade* developed by James Clark (Clark, 1998). This turned out to be extremely cumbersome to implement, and was abandoned. There were a number of commercial products such as *Omnimark Light* (Ominimark Corp; 1998), *TXL* (Legasys Corp; 1998) and *PatML* (IBM Corp; 1998) which looked promising but could not be used since we wanted our transducer to be in the public domain.

We subsequently began to examine available XML transduction resources. XSL (Clark, Deach, 1998) was still not mature enough to rely on as a core for the language. In addition, XSL did not (at the time) provide for rooted, recursive transductions needed to convert the complex data structures found in DicoPro’s lexical

¹DicoPro was a project funded within the Multilingual Information Society programme (MLIS), an EU initiative launched by the European Commission’s DG XIII and the Swiss Federal Office of Education and Science.

²Project participants were: HarperCollins, Hachette Livre, Oxford University Press.

data.

Edinburgh's Language Technology Group had produced a number of useful SGML/XML manipulation tools (LTG, 1999). Unfortunately none of these matched our specific needs. For instance, *sgmltrans* does not permit matching of complex expressions involving elements, text, and attributes. Another LTG tool, *sgprg* is more powerful, but its control files have (in our opinion) a non-intuitive and complex syntax³.

Since a large number of standardized XML APIs had been developed for the Java programming language this appeared to be a promising direction. In addition, Java's portability was a strong drawing point. The API model which best suited our needs was the "Document Object Model" (DOM) with an underlying "Simple API for XML" (SAX) parser.

The event-based SAX parser reads into memory only the elements in the input document relevant to the transformation. In effect, XMLTrans is intended to process lexical entries which are independent of each other and that have a few basic formats. Since only one entry is ever in memory at any given point in time, extremely large files can be processed with low memory overhead.

The DOM API is used in the transformation process to access the the element which is currently in memory. The element is transformed according to rules specified in a rule file. These rules are interpreted by XMLTrans as operations to perform on the data through the DOM API.

We begin with a simple example to illustrate the kinds of transformations performed by XMLTrans. Then we introduce the language concepts and structure of XMLTrans rules and rule files. A comparison of XMLTrans with XSLT will help situate our work with respect to the state-of-the-art in XML data processing.

2 An example transformation

A typical dictionary entry might have a surprisingly complex structure. The various components of the entry: headword, part-of-speech, pronunciation, definitions, translations, may themselves contain complex substructures. For DicoPro, these structures were interpreted in or-

³The LTG have since developed another interesting transformation tool called XMLPerl.

der to construct HTML output for typographical rendition and also to extract indexing information.

A fictitious source entry might be of the form:

```
<entry>
<hw>my word</hw>
<defs>
<def num="1">first def.</def>
<def num="2">second def.</def>
</defs>
</entry>
```

We would like to convert this entry to HTML, extracting the headword for indexing purposes. Applying the rules which are shown in section 4, XMLTrans generates the following output:

```
<HTML>
<!-- INDEX="my word" -->
<HEAD>
<TITLE>my word</TITLE>
</HEAD>
<BODY>
<H1>my word</H1>
<OL>
<LI VALUE="1">first def.</LI>
<LI VALUE="2">second def.</LI>
</OL>
</BODY>
</HTML>
```

If this were an actual dictionary, the XMLTrans transducer would iterate over all the entries in the dictionary, converting each in turn to the output format above.

3 Aspects of the XMLTrans language

Each XMLTrans rule file contains a number of rule sets as described in the next sections. The transducer attempts to match each rule in the set sequentially until either a rule matches or there are no more rules.

The document DTD is not used to check the validity of the input document. Consequently, input documents need not be valid XML, but must still be well-formed to be accepted by the parser.

The rule syntax borrows heavily from that of regular expressions and in so doing it allows for very concise and compact rule specification. As will be seen shortly, many simple rules can be expressed in a single short line.

3.1 Rule Sets

At the top of an XMLTrans rule file at least one “trigger” is required to associate an XML element (e.g. an element containing a dictionary entry) with a collection of rules, called a “rule set”.

The syntax for a “trigger” is as follows:

```
element_name : @ rule_set_name
```

Multiple triggers can be used to allow different kinds of rules to process different kinds of elements. For example:

```
ENTRY : @ normalEntryRules
COMPOUNDENTRY : @ compoundEntryRules
```

The rule set itself is declared with the following syntax:

```
@ [rule set name]
```

For example⁴:

```
@ normalEntryRules
; the rules for this set follow
; the declaration...
```

The rule set is terminated either by the end of the file or with the declaration of another rule set.

3.2 Variables

In XMLTrans rule syntax, variables (prefaced with “\$”) are implicitly declared with their first use. There are two types of variables:

- **Element variables:** created by an assignment of a pattern of elements to a variable. For example: `$a = LI`, where `` is an element. Element variables can contain one or more elements. If a given variable `$a` contains a list of elements `{ A, B, C, ... }`, transforming `$a` will apply the transformation in sequence to `<A>`, ``, `<C>` and so on.
- **Attribute variables:** created by an assignment of a pattern of attributes to a variable. For Example: `LI [$a=TYPE]`, where `TYPE` is a standard XML attribute.

While variables are not strongly typed (i.e. a list of elements is not distinguished from an individual element), attribute variables cannot be used in the place of element variables and vice versa.

⁴XMLTrans comments are preceded by a semicolon.

3.3 Rules

The basic control structure of XMLTrans is the rule, consisting of a left-hand side (LHS) and a right-hand side (RHS) separated by an arrow (“–>”). The LHS is a pattern of XML element(s) to match while the RHS is a specification for a transformation on those elements.

3.3.1 The Left-hand Side

The basic building block of the LHS is the element pattern involving a single element, its attributes and children.

XMLTrans allows for complex regular expressions of elements on the LHS to match over the children of the element being examined. The following rule will match an element `<Z>` which has exactly two children, `<X>` and `<Y>` (in the examples that follow “...” indicates any completion of the rule):

```
Z{ X Y } -> ...;
```

XMLTrans supports the notion of a logical NOT over an element expression. This is represented by the standard “!” symbol. Support for general regular expressions is built into the language grammar: “Y*” will match 0 or more occurrences of the element `<Y>`, “Y+” one or more occurrences, and “Y?” 0 or 1 occurrences.

In order to create rules of greater generality, elements and attributes in the LHS of a rule can be assigned to variables. For instance, we might want to transform a given element `<X>` in a certain way without specifying its children. The following rule would be used in such a case:

```
; Match X with zero or more unspecified
; children.
X{$a*} -> ...;
```

In the rule above, the variable `$a` will be either empty (if `<X>` has no children), a single element (if `<X>` has one child), or a list of elements (if `<X>` has a series of children. Similarly, the pattern `X{$a}` matches an element `<X>` with exactly one child.

If an expression contains complex patterns, it is often useful to assign specific parts to different variables. This allows child nodes to be processed in groups on the LHS, perhaps being re-used several times or reordered. Consider the following rule:

```
Z{ $a = (X Y)* $b = Q } -> ... ;
```

In this case `$a` contains a (possibly empty) list of `<X>`, `<Y>` element pairs. The variable `$b` will contain exactly one `<Q>`. If this pattern cannot be matched the rule will fail.

Attributes may also be assigned to variables. The following three rules demonstrate some possibilities:

```
; Match any X which has an attribute ATT
;
X[ $att = ATT ] -> ...;
```

```
; Match any X which has an attribute
; ATT with the value "VALUE".
;
X[ $att = ATT == "VALUE" ] -> ...;
```

```
; Match any X with an attribute
; which is NOT equal to "VALUE"
;
X[ $att = ATT != "VALUE" ] -> ...;
```

The last type of expressions used on the LHS are string expressions. Strings are considered to be elements in their own right, but they are enclosed in quotes and cannot have attribute patterns like regular elements can. A special syntax, `././`, is used to mean any element which is a string. The following are some sample string matching rules:

```
; Match any string
././ -> ... ;

; Match text "suppress" & newline.
"suppress\n" -> ...;
```

3.3.2 The Right-hand Side

The RHS supplies a construction pattern for the transformed tree node.

A simple rule might be used to replace an element and its contents with some text:

```
X -> "Hello world"
```

For the input `<X>Text</X>`, this rule yields the output string `Hello world`. A more useful rule might strip off the enclosing element using a variable reference on the LHS:

```
$X{$a*} -> $a
```

For the input `<X>Text</X>`, this rule generates the output `Text`. Elements may also be renamed while their contents remain unmodified. The following rule demonstrates this facility:

```
$X{$a*} -> Y{$a}
```

For the input `<X>Text</X>`, the rule yields the output `<Y>Text</Y>`. Note that any children of `<X>` will be reproduced, regardless of whether they are text elements or not.

Attribute variables may also be used in XML-Trans rules. The rule below shows how this is accomplished:

```
X[$a=ATT]{$b*} -> Y[OLDATT=$a]{$b}
```

Given the input `<X ATT="VAL">Text</X>`, the rule yields the output `<Y OLDATT="VAL">Text</Y>`.

Recursion is a fundamental concept used in writing XML/Trans rules. The expression `@set_name(variable_name)` tells the XML-Trans transformer to continue processing on the elements contained in the indicated variable. For instance, `@set1($a)` indicates that the elements contained in the variable `$a` should be processed by the rules in the set `set1`. A special notation `@(variable_name)` is used to tell the transformer to continue processing with the current rule set. Thus, if the current rule set is `set2`, the expression `@($a)` indicates that processing should continue on the elements in `$a` using the rule set `set2`. The following rule demonstrates how transformations can be applied recursively to an element:

```
X{$a*} -> Y{@($a)}
```

```
"Text" -> "txeT"
```

For the input element `<X>Text</X>`, the rule generates the output `<Y>txeT</Y>`. Different rule sets can be accessed as in the following rule file segment:

```
X : set1
```

```
@ set1
```

```
X{$a*} -> Y{@set2($a)}
```

```
"Text" -> "txeT"
```

```
@ set2
```

```
"Text" -> "Nothing"
```

Initially, `set1` is invoked to process the element `<X>`, but then the rule set `set2` is invoked to process its children. Consequently, for the input `<X>Text</X>`, the output is `<Y>Nothing</Y>`.

4 Rules for the example transformation

The transformation of the example in section 2 can be achieved with a few XMLTrans rules. The main rule treats the <entry> element, creating a HTML document from it, and copying the headword to several places. The subsequent rules generate the HTML output from section 2:

```
entry : @ entrySet

@ entrySet
entry{$a=hw $b=defs*}
  -> HTML{"<!-- INDEX=" $a "-->"
          HEAD{TITLE{$a} BODY{H1{$a}
                    @($b)}}
defs{$a=def*} -> OL{@($a)}

def[$att=NUM]{$a*}
  ->LI[VALUE=$att]{$a}
```

5 Comparison with XSLT

The advent of stable versions of XSLT (Clark, 2000) has dramatically changed the landscape of XML transformations, so it is interesting to compare XMLTrans with recent developments with XSLT.

It is evident that the set of transformations described by the XMLTrans transformation language is a subset of those described by XSLT. In addition, XSLT is integrated with XSL allowing the style sheet author to access to the rendering aspects of XSL such as formatting objects.

Unfortunately, it takes some time to learn the syntax of XSL and the various aspects of XSLT, such as XPath specifications. This task may be particularly difficult for those with no prior experience with SGML/XML documents. In contrast, one needs only have a knowledge of regular expressions to begin writing rules with XMLTrans.

6 Conclusion

The XMLTrans transducer was used to successfully convert all the lexical data for the DicoPro project. There were 3 bilingual dictionaries and one monolingual dictionary totalling 140 Mb in total (average size of 20 MB), each requiring its own rule file (and sometimes a rule file for each language pair direction). Original SGML files were preprocessed to provide XMLTrans with pure, well-formed XML input. Inputs were in a variety of XML formats, and the output was

HTML. Rule files had an average of 178 rules, and processing time per dictionary was approximately 1 hour (including pre- and postprocessing steps).

This paper has presented the XMLTrans transduction language. The code is portable and should be executable on any platform for which a Java runtime environment exists. A free version of XMLTrans can be downloaded from⁵: http://issco-www.unige.ch/projects/dicopro_public/XMLTrans/

References

- Bingham, H.:1996 'DSSSL Syntax Summary Index', at <http://www.tiac.net/users/ingham/dssslsyn/index.htm>
- Clark, J.:1998 'Jade - James' DSSSL Engine', at <http://www.jclark.com/jade/>
- Clark, J. Ed.:2000 'XSL Transformations (XSLT) Version 1.0: W3C Recommendation 16 November 1999,' at <http://www.w3.org/TR/1999/REC-xslt-19991116>
- Clark, J. and Deach, S. eds.:1998 'Extensible Stylesheet Language (XSL) Version 1.0 W3C Working Draft 16-December-1998' at <http://www.w3.org/TR/1998/WD-xsl-19981216>
- Glazman, D.:1998 'Simple Tree Transformation Sheets 3', at <http://www.w3.org/TR/NOTE-STTS3>
- IBM Corp.:1999 'IBM/Alphawork's PatML', at <http://www.alphaWorks.ibm.com/tech/patml>
- Language Technology Group:1999 'LT XML version 1.1' at <http://www.ltg.ed.ac.uk/software/xml/index.html>
- Legasys Corp.:1998 'The TXL Source Transformation System', at http://www.qucis.queensu.ca/legasys/TXL_Info/index.html
- Omnimark Corp.:1998 'Omnimark Corporation Home Page', at <http://www.omnimark.com/>

⁵Users will also need Sun's SAX and DOM Java libraries (Java Project X) available from: <http://java.sun.com/products/javaprojectx/index.html>: