Processing XML: a rewriting system approach

Alberto Simões[1] and José João Almeida[2]

[1] Esc. Sup. de Est. Industriais e de Gestão
Instituto Politécnico do Porto
alberto.simoes@eu.ipp.pt
[2] Departamento de Informática
Universidade do Minho
jj@di.uminho.pt

Abstract. Nowadays XML processing is performed using one of two approaches: using the SAX (Simple API for XML) or using the DOM (Document Object Model). While these two approaches are adequate for most cases there are situations where other approaches can make the solution easier to write, read and, therefore, to maintain.
This document presents a rewriting approach for XML documents processing, focusing the tasks of transforming XML documents (into other XML formats or other textual documents) and the task of rewriting other textual formats into XML dialects.
These approaches were validated with some case studies, ranging from an XML authoring tool to a dictionary publishing mechanism.

1 Introduction

Nowadays we can assert that most XML manipulation is done using the Document Object Model (DOM) approach. The Simple API for XML (SAX) approach is also used and is, usually, the best solution for parsing big documents.

This document presents yet another approach specially tailored to transform XML documents into other but similar XML documents, or to transform any kind of textual documents into structured XML documents. We present a `Text::RewriteRules`, a rewriting system written in Perl. While it can be used to rewrite and produce any kind of textual document, in this article we will focus on rewriting XML documents, and producing XML from different kinds of textual documents.

While there are other rewriting systems (say AWK or XSLT) they lack the flexibility of the Perl programming language and, especially, of Perl regular expressions engine. Also, they do not support both rewriting approaches. Finally, XSLT approach is not to rewrite the XML document looking at its syntax, but looking to its tree, making the parsing indispensable.

The next section focuses on `Text::RewriteRules` engine, explaining the algorithms available and how rules are written. Follows a section on XML generation from two different kinds of textual documents and in section 4 we will discuss how to rewrite XML into other XML formats.

May, 2010 - Page. 27–38 ≻ATA 2010 — ISBN: 978-972-99166-9-4

2 Rewriting Engine

Although the approaches described in this article can be implemented using any rewriting system, given it supports the regular expressions we will describe shortly, for our experiments we are using `Text::RewriteRules`, a Perl module for coding rewriting systems. `Text::RewriteRules` derived from previous work, where a text-to-speech application was built as the composition of different rewriting systems [2].

The basic concept of a rewriting system is simple: given a text T and a sequence of pairs $(pattern, action)$, check if there exists a pattern matching the text. Every time it does, the associated action is executed.

The usual substitution operator/function that is available on most languages can be seen as a rewriting system with just one rule. Every time the pattern matches a portion of text, this text is substituted by a string (or the result of invoking a function on the matched text). If we take a sequence of substitutions and force them to be applied until all substitutions can not be done, we have a simple but complete rewrite system.

While this simple solution can be used for some applications it is not flexible enough. On some situations a simple pattern can not be used to check if a substitution can be performed or not. We might want to check a database table or any other resource to decide if the substitution should or not be performed. Also, this database might have the information about what should be used to substitute the matched text.

For this to be possible we need some more information, other than the *pattern* and the *action*: we need a *condition*. This condition should be able to perform any kind of computation it needs, ranging from querying a database to performing a web-query in a search- engine. Thus, our rules will be defined as triples: $(pattern, action, condition)$.

This simple concept can be expanded with extra functionalities like the ability to execute code in the *action* or activate some *action* before any *pattern* is tested.

`Text::RewriteRules` supports two different working mechanisms:

- **Fixed-Point Approach**: apply rules in the order they appear, exhausting the first ones before trying to apply the following ones. The system iterates until no pattern matches the text;
- **Sliding-Window Approach**: apply rules in the order they appear, forcing that the matching text is at the beginning of the string. As soon as a pattern matches, a cursor is placed right after the match, and the next rule should match in the cursor position. If no rule can be applied the cursor advances (a character or a word accordingly with the user needs). This system iterates until the cursor arrives at the end of the string.

2.1 Fixed-Point Approach

The fixed-point approach is the easier to understand but not necessarily the most useful. Its idea is based on a sequence of rules that are applied by order.

The first rules are applied, and following rules are applied only then there is no previous rule that can be applied. It might happen that a rule changes the document in a way that the previous rules can be applied again. If that happens, they will be applied again. This process ends when there is no rule that can be applied (or if a specific rule forces the system to end).

This algorithm is specially useful when we want to substitute all occurrences of some specific pattern to something completely different. As an example, consider the e-mail anonymization for mailing list public archives:

```
RULES anonymize
\w+(\.\w+)*@\w+\.\w+(\.\w+)*==>[[hidden email]]
ENDRULES
```

This simple rule will substitute all occurrences of the specified pattern[3] by the specified text. As this process intends to substitute all occurrences of e-mails, and the replacement text does not match the e-mail rule, we do not have the problem of the rewriting system entering an endless loop.

2.2 Sliding-Window Approach

The sliding-window approach tries to match patterns right after the position of a specific cursor, and after some portion is rewritten, the cursor is put just after that portion. So, the sliding-window approach will never rewrite text that was already rewritten.

When using this approach the user does not need to position the cursor. It is automatically initialized in the beginning of the text, and is automatically placed after the matching portion every time a substitution is made. If no rule can be applied at the cursor position it is automatically moved ahead one character (or one word).

This approach is especially useful when the rewrite rules output can be matched by some other rule that will mess up the document.

As a concrete example, consider a rewriting system that does brute-force translation (translates each word using a bilingual dictionary). After translating the English word '*date*' to the Portuguese word '*data*' we do not want it to be translated again as if it were an English word.

To solve this problem we might add a mark to each translated word, and removed all of them at the end. Another option is to use a sliding-window approach: we translate the word at the cursor position, and move it to the right of the translated word.

Follows an example of how this works[4]

[3] This is a simple pattern that does not cover all e-mail cases, but good enough for illustrating the DSL.

[4] While the underscore character is being used as the cursor in this example, `Text::RewriteRules` uses a special non printable character, not found on normal text.

```
_ latest train
último _ train
último combóio _
```

To write this translator using `Text::RewriteRules` we just need to use a flag when defining the rule, so that `Text::RewriteRules` knows it should use a sliding window.

```
RULES/m translate
(\w+)=e=> $translation{$1} !! exists($translation{$1})
ENDRULES
```

2.3 `Text::RewriteRules` Rules

`Text::RewriteRules` programs can include one or more set of rules. After compilation, each rules set will generate a function that receives a text, rewrites it, and returns the resulting text.

The generated functions can be composed with standard functions, or with other rewriting functions.

There are different kind of rules: simple substitution, substitution with code evaluation, conditional substitutions, and others. In the *fixed-point* behavior, the system will rewrite the document until no rule matches, or a specific rule makes the system exit.

Each rewrite system is enclosed between `RULES` and `ENDRULES` strings, as in the following example:

```
RULES xpto
left hand side pattern ==> right hand side
left hand side pattern =e=> right hand side
left hand side pattern ==> right hand  !!  condition
ENDRULES
```

This block evaluates to a function named `xpto` that is the requested rewrite system. It is possible to pass extra arguments to the RULES block adding them after a slash like:

```
RULES/m xpto
left hand side pattern ==> right hand side
left hand side pattern =e=> right hand side
left hand side pattern ==> right hand  !!  condition
ENDRULES
```

The `/m` argument is used to change the default fixed-point algorithm to the sliding windows one.

2.4 Rule Types

Regarding rules, there are very different kinds. To help understanding examples in the next sections a quick presentation of the more relevant rules follows:

- simple pattern substitution rules are represented by the `==>` arrow. The left side includes a Perl regular expression and the right side includes a string that will be used to replace the match. This string can use Perl variables, both global or captured values. These simple rules can include restrictions of application. These restrictions are added at the end of the rule after `!!`. This condition can use global variables or captured values and is written in common Perl syntax.
- in some situations it is important to be able to evaluate the right side of the rule, calculating the value to be used based on portions of the matched string. Evaluation rules are denoted by a `=e=>` arrow. These rules allow conditions as well.
- there are two special rules to be used just the first time the function is called, or to force the rewrite system to exit. The first is the `=begin=>` rule. It has no left side and the right side is Perl code that changes in some manner the text that will be rewritten. The `=last=>` rule does not have a right hand side. It just exits from the rewrite system as soon as the left hand side regular expression matches.

2.5 Recursive Regular Expressions

What makes `Text::RewriteRules` suitable for rewriting XML, as shown in section 4, is the Perl ability to define recursive regular expressions.

Fortunately, regular expressions are not regular anymore [3]. The old definition of regular expressions and their direct conversion to automata is no longer the rule when talking about scripting languages regular expression engines.

Languages like Perl support extensions that make their *regular expressions* powerful tools, making them comparable to grammars, supporting capturing and referencing, look-ahead, look-behind and recursion:

- regular expressions can define capture zones: pieces of the regular expression that, after matching, will be stored for latter usage;
- regular expressions can define look-ahead or look-behind, making the regular expression to match just if a specific expression is (or not) before or after the matching zone;
- recursion was introduced in the Perl world with Perl 5.10[5] and lets the user to specify a regular expression that depends on itself. As an example, consider the following expression that matches a balanced parenthesis block:[6]

```
my $parens = qr/(\((?:[^()]++|(?-1))*+\))/;
```

[5] About January 2008.

[6] We will not explain the regular expression as that would take too much space. You are invited to read Perl man-page `perlre`.

`Text::RewriteRules` includes a set of engineered (regular) expressions that make the language rewriting task easier:

- `[[:BB:]]`, `[[:PB:]]` and `[[:CBB:]]` match balanced bracketed blocks, balanced parenthesized blocks and balanced curly braces blocks, respectively;
- `[[:XML:]]` and `[[:XML(tag):]]` match well formed XML fragment. The latter forces the root element name.

These expressions do not just match, but also capture. For the balanced pairs it is possible to automatically capture its contents and for the XML fragments it is possible to capture the top-level tag name, as well as the top-level tag contents.

2.6 XML Generation

For simpler XML generation from Perl code we will use `XML::Writer::Simple` [1]. This Perl module allows the usage of a Perl function for each tag in use, making XML elements generation as simple as the invocation of a function. Although this module is not part of the rewriting system it helped reducing the code size and raising code legibility.

3 Rewriting Text into XML

This section presents two different situations where the ability to rewrite textual documents and produce XML was the fastest solution. While the two examples share the approach, they differ on the objective:

- the first case study rewrites a textual format into another TEI (Text Encoding Initiative) format, maintaining the existing structure but also detecting and annotating new information;
- follows the creation of a DSL (Domain Specific Language) [6] for XML authoring. In this case the rewriting system acts like a computer language compiler.

3.1 Annotating a Textual Dictionary

Dicionário-Aberto [5] is a project aiming to transcribe a general language dictionary. The transcribing process was performed by volunteers using a textual syntax, based only in bold and italic mark-up (using asterisks and underscores, respectively), new lines to separate entry senses and empty lines to separate word entries.

This basic syntax was chosen so volunteers (with different degrees of knowledge) could transcribe easily. The drawback is the lack of annotation on the resulting textual document.

One of the best formats to describe general dictionaries using any kind of mark-up is the Text Encoding Initiative XML Schema [7]. As a brief analysis of the schema can show, this format has a rich dictionary structure.

Our challenge was to find a method to transform the few annotated dictionary format into TEI. The solution was to use a rewrite approach, enhancing the textual format step by step.

The rewriting system is too big to be presented completely in this article. It contains more than 40 rules. Briefly, the system acts in this order:

1. different entries are separated using the empty line separator. A start and end tag is glued to the result of processing the contents of that entry. Also, some extra end tags are added, closing the definition and sense;
2. follows a set of rules to detect morphologic properties. These properties are in italic. Therefore, as described earlier, they are bounded in underscores. Unfortunately there is more text in italic. To distinguish them, lists of the used morphologic properties and geographic classifiers were created. These lists were used to rewrite morphologic properties into the corresponding TEI tags;
3. finally there is a bunch of rules to fix the generated XML. For instance, after the morphological information it is needed a definition opening tag. This is performed finding all morphologic information closing tags and checking if they are followed by the definition opening tag.

All this process is performed by rewrite, with regular expressions being matched and rewriting the document contents accordingly. This kind of analysis of the document would be really hard to perform with standard parsing techniques[7].

To help the legibility of the rewriting system the `XML::Writer::Simple` module is used. Instead of defining textually the XML being generated, this module defines automatically functions for each tag. These functions generate the opening and closing tag with the same name (eg. a `gramGrp` function would generate a pair of `gramGrp` opening and closing tags). This module functions also take care of generating empty elements when no content is supplied, and generating correctly tag attributes.

3.2 An XML Authoring Tool

XML was designed to be an exchange format. Its syntax enhances the information structure representation, but it has readability (and human authoring) problems. Manual XML authoring would be easier if we could:

- reduce the size of structural information;
- create abreviation mechanisms for constant parts or parametric macros;
- create include mechanisms;
- support scripting capabilities;

This subsection will discuss how to develop a simple DSL for XML documents authoring.

Although the basic principles are generic, the examples will use HTML dialect, in order to be easier to follow.

[7] Unfortunately and given article page limits no example of rules are shown. The complete conversion script can be downloaded from `https://natura.di.uminho.pt/svn/main/ProjectoDicionario/txt2xml`.

Generic transformations The first type of constructs added to XPL (XML Programming Language, the name of the language we are defining) just change the syntactic sugar for XML tags, from the usual start and end tag to a simple function-oriented syntax.

Basically, instead of writing the full XML tags like

```
<h1>XML programming language</h1>

<ul><li>DSL</li>
    <li>see the <a href='...'> XPL manual</a> </li> </ul>
```

XPL lets the user to write

```
h1{XML programming language}

ul{li{DSL}
   li{see the a{href:{...} XPL manual}}}
```

To implement this syntax using `Text::RewriteRules` we defined the following Perl code:

```
1 my $ID = qr{\w+};                      # Identifier
2
3 while(<>) { print loadit(html($_)) }
4
5 RULES html
6 ($ID)[[:CBB:]]          ==>  <$1>$+{CBB}</$1>
7 <(.*?)>($ID):[[:CBB:]] ==>  <$1 $2='$+{CBB}'>
8 (\\[{}])                =e=> saveit($1)     # protect escaped \{
9 ENDRULES
```

Explanatory notes:

line 3 rewrite the standard input with the `html` rewrite system, and reload the saved portions.
lines 5–9 define the `html` rewrite system (that generates the `html` function).
line 6 expand tags (`a{b}` to `<a>b</b>`);
line 7 treat attributes (transform `<a>at:{b}...` in `<a at='b'>...`);

This rewriting system uses two new functions available in `Text::RewriteRules`:

- `saveit` takes a string and saves it in a symbol table. It returns a special token that will be placed in the text being rewritten. This is specially useful to protect portions of text that would be otherwise rewritten by subsequent rules.
- `loadit` performs the inverse operation, replacing all occurrences of the special tokens by the respective stored string.

Abbreviation mechanisms In order to provide programmability constructs, the rewriting system will use an auxiliary table (named `TAB`) to store user-defined functions.

```
1 RULES html
2 ...
3 ($ID)=[[:CBB:]] =e=> savefunc($1,$+{CBB},1),""
4 ($ID)[[:CBB:]]  =e=> $TAB{$1}->($+{CBB})  !! defined $TAB{$1}
```

Explanatory notes:

line 3 support function definition (`f={...}`).The function `savefunc`, (not presented here) inserts a function definition in the auxiliary symbol table.

line 4 detect function invocations (`f{arg}`) and evaluate them (if the function is defined in the symbol table).

Adding these two rules to the `html` rewrite system it is now possible to define macros like the following ones:

```
r   ={font{color:{red} #1}}                  # red text
q   ={div{class:{boxed} #1}}                 # to use CSS box
jpgi={img{src:{#1.jpg} alt:{image of a #1}}} # jpeg image

q{ p{r{Animals:} jpgi{cat} jpgi{donkey}} }
```

Note that `#1` is the argument for the macro, idea stollen from TEX. After processing this document the resulting XHTML document will look like:

```
<div class='boxed'>
  <p>
      <font color='red'> Animals:</font>
      <img src='cat.jpg' alt='image of a cat'></img>
      <img src='donkey.jpg' alt='image of a donkey'></img></li>
  </p>
</div>
```

Include mechanisms Finally, the language also supports different types of include mechanisms:

- split the code in order to reduce the size of the source document and have document modularity (just like `#include` in the C programming language);
- to separate different concepts (example: store function definitions in a *new notation* block;
- to include verbatim examples of code (like \`verbatiminput` in TEX);

For XPL we defined two include mechanisms: `inc` – include, and `vinc` – verbatim include. `inc` provides modularity support:

```
RULES html
...
inc[[:CBB:]]  =e=> ‘cat $+{CBB}‘          !! -f $+{CBB}
inc[[:CBB:]]  =e=> die("can’t open file") !! not -f $+{CBB}
```

The inclusion is just the substitution of the include command by the file contents. The rewriting system will take care to process the new commands in next iterations.

For including verbatim files we defined a second rewrite system, that will replace this verbatim include command by the file contents. This could not be done at the `html` function level as other rewrite rules would rewrite the file contents. The inclusion also needs to protect special characters for HTML inclusion.

```
1  while(<>){ print loadit(verbatim(html($_))) }
2
3  RULES/m verbatim
4  <vinc>(.*?)</vinc>=e=> bpre( protect( ‘cat $1‘ ))!! -f $1
5  ENDRULES
6
7  RULES/m protect
8  <==>\&lt;
9  >==>\&gt;
10 &==>\&amp;
11 ENDRULES
```

Explanatory notes:

line 4 `protect` is rewriting the contents of the file being included.
lines 8–10 transforms HTML special characters into entities.

Scripting embbeding XPL also supports Perl code embbeding, using the `perl` command. Its implementation is just the code execution, and substitution by the respective result string.

```
RULES html
perl[[:CBB:]]=e=> do($+{CBB}),$@ !! -f $+{CBB}
...
ENDRULES
```

4 Rewriting XML into XML

`Text::RewriteRules` can easily rewrite XML documents both as if they were plain text documents or using the defined recursive regular expressions for XML.

These regular expressions are tailored so that they match well formed XML blocks making it easier to remove or replace complete elements subtrees.

Textual XML rewriting is not necessarily faster or more efficient than DOM oriented processing. It all depends on the document size and the DOM structure. But textual XML rewriting can be easier to understand and can be more powerful on some specific situations. Also, these two approaches are not mutually exclusive. One can process the document using the rewrite approach to detect the relevant elements to process, and use a DOM aware tool to process that XML fragment. This is the approach we will show in the next example.

The example will remove duplicate entries from a TMX (Translation Memory eXchange) file. These files size is, usually, quite large, but the structure is quite simple and repetitive. They consist of a list of translation units: pairs of sentences, in two different languages.

```
RULES/m duplicates
([[:XML(tu):]])==>!!duplicate($1)
ENDRULES

sub duplicate {
  my $tu = shift;
  my $tumd5 = md5(dtstring($tu, -default => $c));
  return 1 if exists $visited{$tumd5};
  $visited{$tumd5}++
  return 0;
}
```

In the example the rewritting system is very simple. It matches XML trees that have as the tag `tu` as root element, and substitutes it by nothing in case of it is duplicate. The duplication mechanism processes the translation unit using a DOM approach (using `XML::DT` module [4]), concatenating the translation unit contents, and calculating its MD5. If it is already visited a true value is returned. If not, that MD5 is saved for future reference.

5 Conclusions

The presented case studies shown that the text rewriting approach is a powerful technique to convert textual document formats. Also, the ability to use a module to generate XML tags easily and the availability of recursive regular expressions matching complete well formed XML structures make `Text::RewriteRules` suitable to generate XML and to rewrite XML documents.

The described tools are being used in production in different projects, from the generation of a 30 MB dictionary XML file from a 13 MB text file, taking about nine minutes. The difference on the file sizes show the high number of generated XML tags.

We also described how this rewriting approach can be used for the creation of domain specific languages, namely for the authoring of XML documents.

Finally, the ability to rewrite XML documents in other formats (or to rewrite contents) make the tool suitable for rewriting XML dialects. This approach does not require the creating of a complex data structure in memory, just loading the document as text, and searching on it for the relevant fragments. These can be extracted and processed by a common DOM-oriented approach. Using a streaming library this means it is possible to process huge files where small chunks need processing without loading the full document to memory.

All the modules and tools described are available from CPAN[8] and can be used free of charge.

References

1. José João Almeida and Alberto Simões. Geração dinâmica de APIs Perl para criação de XML. In José Carlos Ramalho, Alberto Simões, and João Correia Lopes, editors, *XATA 2006 — 4ª Conferência Nacional em XML, Aplicações e Tecnologias Aplicadas*, pages 307–314, Portalegre, February 2006.
2. José João Almeida and Alberto Manuel Simões. Text to speech — *"A rewriting system approach". Procesamiento del Lenguaje Natural*, 27:247–253, 2001.
3. Jeffrey E. F. Friedl. *Mastering Regular Expressions.* O'Reilly & Associates, Sebastopol, California, January 1997.
4. Alberto Simões. Down translating XML. *The Perl Review*, 1(1):6–11, Winter 2004.
5. Alberto Simões and Rita Farinha. Dicionário aberto: Um novo recurso para PLN. *Vice-Versa*, 2010. forthcomming.
6. Arie van Deursen, P. Klint, and J.M.W. Visser. Domain-specific languages. Technical Report SEN-R0032, ISSN 1386-369X, CWI, 2000. an annotated bibligraphy.
7. Edward Vanhoutte. An introduction to the TEI and the TEI Consortium. *Lit Linguist Computing*, 19(1):9–16, April 2004.

[8] Comprehensive Perl Archive Network, available at `http://search.cpan.org/`