Exploring and Extracting Nodes from Large XML Files

Guy Lapalme

January 2010

Abstract

This article shows how to deal simply with large XML files that cannot be read as a whole in memory and for which the usual XML exploration and extraction mechanisms cannot work or are very inefficient in processing time. We define the notion of a skeleton document that is maintained as the file is read using a pull-parser. It is used for showing the structure of the document and for selecting parts of it.

1 Introduction

XML has been developed to facilitate the annotation of information to be shared between computer systems. Because it is intended to be easily generated and parsed by computer systems on all platforms, its format is based on character streams rather than internal binary ones. Being character-based, it also has the nice property of being readable and editable by humans using standard text editors.

XML is based on a uniform, simple and yet powerful model of data organization: the generalized tree. Such a tree is defined as either a single element or an element having other trees as its sub-elements called children. This is the same model as the one chosen for the Lisp programming language 50 years ago. This hierarchical model is very simple and allows a simple annotation of the data.

The left part of Figure 1 shows a very small XML file illustrating the basic notation: an arbitrary name between `<` and `>` symbols is given to a node of a tree. This is called a *start-tag*. Everything up until a corresponding *end-tag* (the same tag except that it starts with `</`) forms the content of the node, which can itself be a tree. Such node in the tree (`r`, `a`, `b` and `c` in Figure 1) are called *elements*. Elements can also contain character data (`c` in Figure 1) and even mix character data and elements (`b` in Figure 1). An XML element with no content can be indicated with an end-tag immediately following a start-tag and can be abridged as an empty-element tag: a start-tag with a terminating `/` (`a` in Figure 1). Additional information can be added to a start-tag with attribute pairs comprising the name of the attribute (`id` in tags `a`, `b` and `c` in Figure 1), an equal sign and the corresponding character string value within quotes (e.g. `"01"`). Attributes can also be added to an empty element (`a` in Figure 1).

```
<r>
    <a id="i1"/>
    <b id="i2">
        <c id="i3">e</c>d<c id="i4">f</c>
    </b>
</r>
```

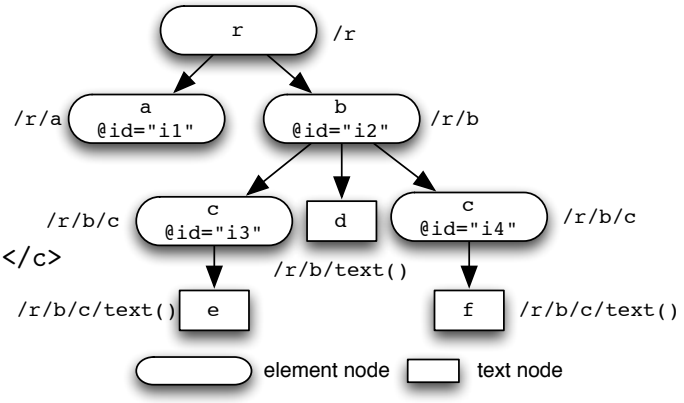

Figure 1: Small XML file on the left with, on the right, the corresponding tree structure and the XPath expressions describing the access to each node.

The right part of Figure 1 shows that this notation is equivalent to a tree data structure where each node is labelled with its name and attributes. Character data appears as leaf nodes. An empty element is a node with no sub-tree.

XML is widely used in computing systems to systematize structured data as an alternative to databases. Many relational databases also offer XML specific features for indexing and searching. Because of the portability of its encoding and the fact that XML parsers are freely available, it is also used for many tasks requiring flexible data manipulation to transfer data between systems, as configuration files of programs and for keeping information about other files.

XML has the (well deserved) reputation of being verbose but it must be kept in mind that this notation is primarily aimed at communication between machines for which verbosity is not a problem but uniformity of notation is a real asset.

However the size of XML files can grow fast (some cases, they can be many megabytes long) and this complicates the access to their content (i.e. identification and output of a few nodes). XML file exploration and extraction tools rely on XPath [5] processors that evaluate a tree oriented expression over the global tree structure to select nodes in the tree. In order to evaluate such an expression, the XPath processor first reads the file and builds, in memory, an internal tree structure over which the expression is evaluated. This is not a problem with small files (less than a megabyte or two) but it becomes problematic for larger ones because, in memory, the tree structure is much larger than the original text file.

2 Related work

The problem of transforming large XML files has long been recognized and the best attempt to deal with it is STX [7] which is an XML-based language for transforming XML documents into other XML documents without building the full tree in memory. It defines a new interrogation and transformation language adding some restrictions to the standard XPath [5] and XSLT [12]. These modifications are aimed at guaranteeing that the transformation templates can be applied in streaming mode over events that are

received sequentially from an XML parser. It relies on a data model keeping in memory at a given time a single node of the tree and a stack of its parents up to the root. *Joost* [4] is a prototype implementation of STX in Java and `XML::STX` [6] is a Perl implementation. Our approach relies on a similar representation but it is more modest in goal: we do not aim at transforming an XML document but we rather restrict ourselves to the extraction of a small part of it. The extraction mechanism is a single XPath expression with small constraints that allow the programmer an intuitive control on the amount of memory needed to process the file.

XStream [11] is another approach for the efficient streaming transformation of XML files using a declarative functional language based on OCAML which is quite different from the usual XPath language. Our approach does not imply learning a new formalism other than the original XPath but, we do not try to transform the original large file, only to extract a small part.

A different approach to the extraction of nodes from large XML files is *projection* [15] applied in the context of XQuery. The system analyzes the query and then parses the XML file but only keeps in memory the *relevant* parts of the document to ensure that the result of the query is the same in the projected document as in the original document. This analysis is relatively delicate and has been implemented in O'CAML in the Galax XQuery processor. Saxon (Entreprise Edition) also supports this capability in its XQuery implementation. Our approach can be considered a simplified projection which does not create a projected document but which outputs on the fly only the projected part of the document corresponding to an XPath expression.

Another interesting approach to dealing with large XML files is the *Gadget* XML inspector [16] that produces an index of all XPath expressions for locating nodes in a file. That index is kept in a database then used for a faster access to the content of the XML file. We use a similar idea to explore the XML file but our approach is much simpler because we do not keep track of these expressions but we merely write them out and let other tools deal with this textual output.

3 Context of use

In order to extract small portions of large text files, we often resort to sequential reading of each line and decide for each if it is to be kept or not. A well-known utility for this operation is `grep` which reads a file line by line and copies only the ones matching a given regular expression. Our goal is to find a similar type of utility as `grep` but for operating on XML files.

In the case of tree-oriented XML files, `grep` is not very useful because the regular expression cannot take into account the embedded structure of the tags. And worse, in many case, XML files are composed of only one line (to save space...) because end-of-lines do not have any special meaning except within character content. It is the tree structure that matters and it must be taken into account in the extraction process.

In this article we propose an *equivalent of a line* in a tree-structured XML file in order to be able to process the file sequentially by evaluating a somewhat restricted, but nevertheless useful, form of XPath expression for selecting some parts of an XML document without having to build the tree in memory before evaluation. There is no need

```
/r                  | 2 /r/b/c          | /r
/r/a                | 2 /r/b/c/text()   | /r/a[@id="i1"]
/r/b                | 1 /r              | /r/b[@id="i2"]
/r/b/c              | 1 /r/a            | /r/b[@id="i2"]/c[@id="i3"]
/r/b/c/text()       | 1 /r/b            | /r/b[@id="i2"]/c[@id="i3"]/text()[.="e"]
/r/b/text()         | 1 /r/b/text()     | /r/b[@id="i2"]/text()[.="d"]
/r/b/c              |                   | /r/b[@id="i2"]/c[@id="i4"]
/r/b/c/text()       |                   | /r/b[@id="i2"]/c[@id="i4"]/text()[.="f"]
```

Figure 2: XPath expressions describing the content of Figure 1: the left part shows all XPath expressions as they are encountered during reading of the file; the center part, displays the count of each distinct Xpath expression and the right part also shows the values of the attributes and the text nodes.

to learn a new language like in XStream or STX. The implementation that we describe is relatively simple with a few hundred lines of Java code built on the standard XML packages already part of the version 6 of the Java™ Platform, Standard Edition.

We propose to use a series of descriptions of a *skeleton document* built by describing the path from the root to a node of the tree; this is a simplification of the data-model of STX [7]. Figure 1 shows the description of each element below or beside each node.

In the left column of Figure 2, we can see the list of XPath expressions identifying the nodes of the tree from the right-hand part of Figure 1. Each line is the XPath expression that can be used to access a node in the XML document. A single expression can return more than one node (see `/r/b/c`). In section 4, we show how these XPath expressions can be built sequentially by reading the XML with a Pull Parser which is a space-efficient way of processing XML files while keeping only a single *spine* going from the root to a leaf.

As for `Gadget` [16], these XPath expressions can used to understand the structure of an XML document and this line-delimited output can be combined with other Unix tools. For example, piping the content of the first column of Figure 2 to

```
sort | uniq -c | sort -n -r -s+
```

we produce the content shown in the middle column of Figure 2 which shows the count of the number of different skeleton documents and displays them in decreasing order of frequency and in alphabetical order within equal counts (stable sort with the `-s` argument on the second sort)[1].

It can also be interesting to describe the attributes and their values and the values of the text nodes which is shown in the rightmost column of Figure 2. Each node name is followed by a *predicate*, a boolean expression testing an expression on the current node. In this case, it is simplified for describing the value of an attribute with its string value: the name of the attribute is preceded by `@` and followed by an `=` and the string giving its value. There are as many predicates as attributes for a given node. In the case of a text node, the predicate is of the form `.=` because, in XPath, the period is the value of the

[1]As this type of display is often useful for getting an idea of an XML file, our program can also compute this display directly with a single command argument

```
<root>
<c id="i3">e</c>
<c id="i4">f</c>
</root>
```

Figure 3: XML document produced by evaluating the XPath expression `/r/b/c` in the XML content Figure 1.

current node. We could have devised a simpler form to show the values but we wanted them to be legal (and systematic) XPath expressions.

Once we know the structure of an XML file, one of these XPath expressions can be used for efficiently extracting nodes, by incrementally evaluating it on the skeleton documents built using a similar process as the one used for displaying the XPath expressions. Figure 3 shows the result of evaluating `/r/b/c` on the content of Figure 1. It is a well-formed XML document containing the nodes of the file matching the XPath expression. The name of the root element can be customized by a command line argument and an XSLT stylesheet can be further applied to each extracted node.

This allows the extraction of parts of an XML file using space-efficient XPath expressions that are evaluated only on skeleton documents. XPath expressions are not limited to the ones generated by the exploration mechanism but they must start at the root not make any references to nodes not in a skeleton document such as siblings or descendants. The hierarchical structure of the XPath expression often allows the skipping of large parts of a document without having to evaluate all skeleton documents. For example in Figure 1, when extracting `/r/a`, the subtree rooted at `b`, can be safely ignored because it will never match. The corresponding part of the file will of course be read but the corresponding skeleton document need not be built in full.

Section 4 describes how to build skeleton documents in a single pass over the XML file. Section 5 shows how to extract the parts of the file that match an XPath expression without having to keep the entire XML file in memory at any given time. The user can control the memory requirements via the form of the XPath expression used for extraction. Section 6 describes the implementation of this idea in Java, in Ruby and its adaptation in Joost. Section 7 presents some execution times on large XML files.

4 Exploring the XML file

We now show how to build the set of skeleton documents using a one-pass XML parsing[2]. Pull parsing [14, Sec 6.3][17][2], available since Java 1.6 as *Stream API for XML* (StAX), gives the programmer the control when the reading of an XML file through on the notion of a cursor that walks the document from beginning to end. The cursor can only move

[2]We have implemented this methodology using both a *pull-parser* and *SAX parser* which have shown similar performance in both time and memory. In this paper, we describe the *pull-parser* approach as it seems simpler to explain because the state can be conveniently kept in recursive calls. Global variables have to be used between call-back methods to decide if some elements must be skipped or kept in memory for further testing. For the output using a pull-parser, we also use the Stream Approach to XML so that we do not have to build the document in memory when we skip some parts.

Algorithm 1 Explore an XML file using a skeleton document. Comments and processing-instruction nodes that cannot have children are dealt similarly to text nodes but are not described explicitly here.

```
while there are tokens left in the XML file do
    token ← next token
    if token is a start-tag then
        convert token to an element node;
        add it as child of the bottom node of the skeleton document;
        output the XPath expression describing the leaf of the document
    else if token is a character-data then
        convert token to a text-node;
        add it as child of the bottom node of the skeleton document;
        output the XPath expression describing the leaf of this document;
        remove this character-element from the skeleton document
    else if token is an end-tag then
        remove the bottom node of the skeleton document
    else if token is a whitespace only element then
        skip it
```

forward and it points to a single point in the XML file. Only a small part of the whole XML document needs to reside in memory at any single time, so this approach is quite memory-efficient. Because the programmer controls which methods are called when a given token is at the cursor, it is possible to keep contextual information when dealing with a token. A token can be a start-tag with its attributes, an end-tag, character content (containing possibly only white-space), a processing-element, a comment, etc...

To display the set of skeleton documents, we read each token and create a DOM structure (i.e. element nodes with a single child or a text node) when going forward in the XML file. The skeleton document is built incrementally as the XML file is read token by token as shown in Algorithm 1.

For outputting an XPath expression, the skeleton document is traversed from the root to a leaf node and we output a `/` followed by the name of the node (possibly qualified by an appropriate namespace) until the bottom node is encountered. In the case of a text node, then the node-test `text()` is output. When attributes and values are requested, then we add appropriate predicates of the form `[.="value"]` after the name of the node or the node-test.

The memory requirements for this pass depend on the longest path length going from the root to a leaf of the XML document. As data-oriented XML files are most often much more *wide* than *deep*, it is not a problem to deal with huge XML files with relatively modest memory requirements. Each XPath expression can be seen as a thread from the root element to a leaf node of the XML document. We consider this type of XPath expression as the equivalent of a *line* or a *record* in other stream oriented processing as promoted by the many utility programs available in Unix like systems.

5 Extracting nodes from the XML file

Extracting nodes from an XML file is only a matter of evaluating an XPath (or XQuery) expression over a document. Given the simple form of extraction we deal with, we will limit ourselves to the XPath syntax. An XPath expression is composed of steps separated by /. Each step designates an axis which can be understood as the direction in which the nodes will be collected from the current node (called here the starting node) in the document tree. An axis is a predefined name followed by :: (e.g. child, parent, attribute). In this document, we use only the `child::` axis (which is simply omitted in the abbreviated syntax) but the full XPath syntax is supported subject to the limitations described below. The axis specification is followed by a node test which is often the name of a node or * to indicate any element. The node(s) so specified can be further restricted with a predicate expression within square brackets; this is a boolean expression evaluated at each node, which is filtered out when it evaluates to false.

Evaluating an *XPath expression* over a document to extract a few nodes corresponds to the following XSLT transformation:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
     <root>
       <xsl:copy-of select="Xpath expression"/>
     </root>
  </xsl:template>
</xsl:stylesheet>
```

or to the following XQuery expression:

```
<root>
   {Xpath expression}
</root>
```

For small XML files, an XPath expression is easily evaluated because the XML processor reads all the file and builds an internal tree structure on which the XPath expression is evaluated to return the list of nodes matching this expression. Unfortunately, this model does not work on large XML files because the memory requirements for building the internal data structure of the whole document can exceed the memory available to the program.

When only a few nodes need to be extracted from an XML file, it can be quite time consuming to build the internal structure for a whole file only to keep a few nodes after evaluation. Ideally we would like to evaluate the XPath expression on a dynamically built structure as the file is read. To achieve this, we propose to limit the evaluation of the XPath expression on skeleton documents which, as we have shown in the previous section, can be built incrementally in a single pass over the XML file.

In order for the evaluation of an XPath expression over the skeleton document to give the same results as its evaluation over the whole document, the allowed XPath expression language is limited somewhat with the following restrictions:

- the XPath expression should return a list of nodes (possibly a single one) but not a number, nor a boolean nor a string, but it can return a text node;

XPath expression	Parts

`/r/b/c`	`/r`
	`/r/b`
	`/r/b/c`

`/r/b[substring(@id,2)>"1"]/c`	`/r` `/r/b[substring(@id,2)>"1"]/c`

Table 1: Splitting of an XPath expression into parts.

- the expression should start at the root (i.e. with a `/`) and should indicate each level of the tree, no `//` or `descendant::` axis; usually each level indicates the name of a node to match but it can also be a general node-test such as `*` or `node()`;
- the predicate expression should not refer to siblings or to functions that refer to a position within siblings or to the number of siblings; the predicate must return a boolean value and not an integer as it is allowed in the full XPath language;
- the predicate expression cannot use any XSLT variable (preceded by `$`).

In practice, for the purpose of extracting nodes from an XML file, these limitations are not very problematic. The extraction language is thus similar to the regular expression language used by tools such as `grep`. In our implementation, each extracted node is further transformed with an XSL stylesheet (the identity transformation if none is specified) in which the full XPath transformation language is available to format or further transform the extracted node.

To improve the efficiency of the XPath expression evaluation, we use the tree structure of the document and the XPath expression by matching incrementally each level in the tree. When an expression at a high level does not match, then lower levels will never match and thus the parser can skip parts of the file without building the corresponding internal tree structure and evaluating lower-level expressions.

The XPath expression is split into *parts* that will be evaluated as the skeleton document is built by pulling tokens from the file. The parts correspond to the steps of an XPath expression until one step contains a predicate. Because the evaluation of a predicate cannot predict its values for parts further down the tree, this last step contains the rest of the XPath expression. Table 1 gives some examples of this splitting process.

The skeleton document over which the XPath expression is evaluated is built similarly to the process we described in the previous section but by evaluating each partial XPath on the skeleton document as it is built. Should an intermediary (not the last one) XPath expression evaluate to false, the XML parser skips over the rest of this element and its internal node without building the skeleton document or evaluating the XPath expression.

Algorithm 2 shows the XPath expression process as it moves forward in the XML in the XML file.

Algorithm 2 Extract nodes from an XML file by evaluating an XPath expression separated in n parts

```
while there are tokens left in the XML file do
    token ← next token;
    i = 0
    if token is a start-tag or character-data then
        if i < n − 1 then
            convert token to a node (element or text node);
            add this node at the bottom of the skeleton document;
            if part i of XPath matches the skeleton document then
                i = i + 1;
                continue the while loop;
            else {no match}
                if token is an start-tag then
                    skip the rest of this element (including embedded elements)
        else {i = n − 1}
            if token is an start-tag then
                read the rest of the current element;
                create an element node with its content
            else {token is character-data}
                convert the character-data to a text-node
            add the created node at the bottom of the skeleton document;
            if the last part of the XPath matches the skeleton document then
                output the node as result
        remove the bottom element of the skeleton document
    else if token is a end-tag then
        i = i − 1;
        remove the bottom element of the skeleton document
```

6 Implementation

Algorithms 1 and 2 can easily be implemented in any programming language providing a pull-parser, as well as classes for manipulating XML document nodes and element and tools for evaluating XPath expressions over a document built in memory. This section gives an overview of the tools we use for implementing them in Java and in Ruby. Since both algorithms are based on the output of a pull parser to build the skeleton document, this code can be shared among the two algorithms.

6.1 Implementation in Java

To read the XML document using the pull parsing approach, we use the classes and interfaces from the `javax.xml.stream` package. The internal tree representation of the skeleton document is built using classes from `org.w3c.dom`. `javax.xml.xpath` provides an API for the evaluation of XPath expressions and access to the evaluation environment and `javax.xml.transform` defines APIs for performing a transformation from source to result.

Combining these tools from the *Java™ Platform, Standard Edition 6* [1] , allows an implementation in a few hundred lines of Java code[3]. The XPath expression that will be evaluated for each skeleton document and the transformation to be applied to each extracted node can be *compiled* once when the program starts and so the execution is relatively fast (a few seconds for dealing with a 20 megabytes XML file). Execution speed depends on the form of the XPath expression and whether large portion of the file can be skipped.

Using the Java SE 6 allows the use of the XPath 1.0 [9] and XSLT 1.0 language [8] but adding the Saxon jar file [13] in the class path allows the use of XPath 2.0 [5] and XSLT 2.0 [12] without changing the Java source file.

6.2 Implementation in Ruby

In Ruby [3], we use the `REXML`, a pure Ruby non-validating XML toolkit that provides tree, stream and pull APIs and which includes an XPath 1.0 implementation. Since Ruby 1.8, REXML is included in the standard Ruby distribution. Unfortunately, it does not provide a transformation API so the extracted node are output verbatim from the source. Given the usual terseness of Ruby for expressing, the implementation is less than 200 lines long but because it is a script language, it is much slower than the Java program.

6.3 Implementation with Joost

As discussed in section 2, Joost is a Java implementation of a streaming transformation (STX) engine that is applied on a structure similar to the skeleton document we have described in this paper. Using a transformation template matching `node()` and dumping the name and attributes of the ancestor nodes (available with the XPath expression `//*`), we can produce the same output as the exploration mode's.

To extract nodes corresponding to a given XPath expression, we generate a template containing as `match` attribute the content of the XPath expression and output its content. Some care has to be taken, to make sure that all nodes and attributes are traversed.

So we see that this idea can be implemented in different ways depending on the XML tools available.

7 Testing

In order to assess to what extent this approach is practical with real XML data, we ran the implementations described in the previous section on three relatively large XML files describing dictionary entries and one widely used XML benchmark:

Dubois classes and other linguistic and semantic information about more than 25 000 French verbs. More details at `http://rali.iro.umontreal.ca/Dubois`;

Morphalou more than 500 000 French word forms. More details at `http://www.cnrtl.fr/lexiques/morphalou/`, Version 1.0 was used in these experiments;

[3]The Java source code and its documentation are available at `http://www.iro.umontreal.ca/~lapalme/ExamineXML`

	Dubois	**Morphalou**	**GCIDE**	**XMark**
# chars	23 430 946	80 512 282	63 511 211	116 524 435
# words	1 441	3 179 427	5 732 917	12977 565
# lines	540 721	794 285	1 179 297	2035 122
# entries	25 609	67 376	130 590	-

Table 2: Characteristics of the three large XML files used for testing the implementations reported in Table 3. The number of words, lines and lines were determined using the Unix `wc` program. The number of *entries* indicates the *natural* units of each dictionary and was determined using appropriate XPath expressions shown in Table 3. This measure is not relevant for XMark

GCIDE An XML version of the `Gnu version of the Collaborative International Dictionary of English`. More details at `http://rali.iro.umontreal.ca/GCIDE`.

XMark An XML benchmark created with a random generator to exercise the management of large volumes of XML data. More details at `http://www.xml-benchmark.org/`. The original benchmark was developed for XQuery but has been adapted to XPath by Massimo Franschet [10] who designed a set of expressions to test both functional and performance aspects of XML processors.

Some characteristics of those dictionaries are given in table 2.

Although these tests are rudimentary and surely not *exhaustive*, they make the point that building a skeleton document can be efficient enough to extract information from large XML files.

XPath expression	# of results	Examine	Joost	REXML
Dubois				
`/dubois/verbe/mot`	25 609	5.7	6.3	352.3
`/dubois/verbe/mot [matches(.,'.*voir$')]`[a]	140	6.4	7.9	479.4
`/dubois/verbe/mot [matches(.,'.*voir$')]/mot`	140	5.7	9.8	211.8
`/dubois/verbe/mot [matches(.,'.*voir$')]/text()`	140	6.6	10.4	7.8
Morphalou				
`/lexicalDatabase/ lexicalEntry/inflectionGroup`	67 376	7.7	13.9	
`/lexicalDatabase/... /@orthography`[b]	7 321	14.0	15.3	
GCIDE				
`/dictionary/body/ entries/entry/hw`	130 590	10.9	16.1	
XMark				
`/site/closed_auctions/closed_auction/ annotation/description/text/keyword`[c]	4 042	6.4	18.1	
`/site/people/person[...]/name`[d] `(creditcard or profile)]/name`	7 181	5.5	18.6	

Table 3: Time in seconds, on a 1.6 GHz MacBook running MacOSX 10.6.2, for extracting entries corresponding to the XPath expression given in the first column. The Saxon 9 HE XSL library was used in these tests. As the time needed for the REXML implementation were so long compared to the other two, we did not run all tests with it. In no case, did we ever run out of memory, even though none of these files could be read using a *standard* XPath processor.

[a] Show all French verbs that end with the letters `voir`, such as `apercevoir, recevoir, etc.`

[b] The full XPath expression is `/lexicalDatabase/lexicalEntry/inflectionGroup/inflection [@grammaticalMood='indicative' and @grammaticalNumber='singular' and @grammaticalPerson='thirdPerson' and @grammaticalTense='simplePast']/ @orthography` which displays all French verbs conjugated at the 3rd person singular at the indicative simple past.

[c] This expression, numbered A1 in [10], corresponds to "keywords in annotations of closed auctions"

[d] The full XPath expression, numbered A8 in [10], `/site/people/person[address and (phone or homepage) and (creditcard or profile)]/name` which corresponds to "people that have declared address and either phone or homepage and either credit card or profile".

8 Conclusion

In this paper, we have described a simple approach for exploring the content of an XML file and manipulating its content using line-oriented tools. We have also described a `grep`-like tool for selecting nodes in an XML file and producing a new XML file but not needing to keep the whole file in memory. This method can be implemented in a few hundred lines of code and does not need to rely on sophisticated databases or a new formalism and interpreter.

9 Acknowledgment

This work was performed during a stay at the Laboratoire d'Informatique Fondamentale at the Université de Marseille. Thanks for Michael Zock and Paul Sabatier for many fruitful discussions.

References

[1] Java™ Platform, Standard Edition 6 API Specification. `http://java.sun.com/javase/6/docs/api/`, 2008.

[2] XML Pull Parsing. `http://www.xmlpull.org/`.

[3] The Ruby Language. `http://www.ruby-lang.org/`, 2008.

[4] Oliver Becker. Joost. `http://joost.sourceforge.net/`, 2009.

[5] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robin, and Jérôme Siméon. XML Path Language (XPath) 2.0. `http://www.w3.org/TR/xpath20`, January 2007.

[6] Petr Cimprich. XML::STX. `http://search.cpan.org/dist/XML-STX/`, Dec 2004.

[7] Petr Cimprich, Oliver Becker, Christian Nentwich, Honza Jiroušek, Manos Batsis, Paul Brown, and Michael Kay. Streaming transformations for XML (STX). Technical report, `http://stx.sourceforge.net/documents/spec-stx-20070427.html`, April 2007.

[8] James Clark. XSL Transformations (XSLT) Version 1.0. `http://www.w3.org/TR/xslt`, November 1999.

[9] James Clark and Steve DeRose. XML Path Language (XPath) 1.0. `http://www.w3.org/TR/xpath`, November 1999.

[10] Massimo Fransceschet. XPathMark. `http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/index.html`, 2009.

[11] Alain Frisch. Xstream. `http://yquem.inria.fr/~frisch/xstream/`, 2006.

[12] Michael Kay. XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20/, January 2007.

[13] Michael Kay. SAXON, The XSLT and XQuery Processor. http://saxon.sourceforge.net/, August 2009.

[14] Guy Lapalme. Looking at the forest instead of the trees. XML tutorial available at http://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTheTrees/.

[15] Amélie Marian and Jérôme Siméon. Projecting XML documents. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB '03)*, Berlin, Germany, 2003.

[16] Stefano Mazzocchi. Gadget. http://simile.mit.edu/wiki/Gadget, May 2006.

[17] Sun Corp. The Java Web Services Tutorial. Available at http://java.sun.com/webservices/docs/1.6/tutorial/doc/.