DOM-like XML Parsing Providing Import and Export with Bounded Resources

Pedro Côrte-Real[1] pedro@pedrocr.net
Gabriel David[1,2] gtd@fe.up.pt
Silvestre Lacerda[3] lacerda@iantt.pt

[1] Faculdade de Engenharia da Universidade do Porto
[2] INESC – Porto
[3] Instituto dos Arquivos Nacionais/Torre do Tombo

Abstract. XML parsing is usually done using DOM or SAX. DOM is memory-bound while SAX provides a fairly low level of abstraction. We propose an approach to create an object-oriented mapping of an XML format. Based on the resulting representation, API's for reading and writing complete documents in a DOM-like way are provided. To handle very large documents we propose an event-based API that returns full elements as events and a second API to create unlimited sized documents with memory usage proportional to the maximum tree depth.

1 Introduction

The most common models for XML interpretation and generation are DOM[3] and SAX[2]. DOM gives you access to the whole document at once by keeping it all in memory at the same time. This makes the memory usage grow linearly with the document size. SAX solves this limitation by allowing the instrumentation of parser events but is a bit cumbersome to program since the events are fine grained requiring applications to handle them at a low level of abstraction. StAX[6] is analogous to SAX but provides a pull-style API although it still does a very fine grained parsing of XML. Because of this, programmers tend to end up using DOM as long as it is feasible and switching to SAX or StAX when the datasets become too large. This results in a complete change in the way to access the same XML format.

The solution proposed to the problem of XML parsing relies on first creating a mapping of a XML format to an object-oriented structure. Similar approaches have been adopted before[14,5,7]. The proposed solution is more lightweight without any code generation or automatic conversion from DTD's or XML Schemas[4].

API's using the object mapping are provided to import and export documents as a whole, as would be done with DOM. An event parser where events are complete elements mapped to the same objects is provided to allow the importing of unlimited sized documents. A way to export unlimited-sized documents is also provided by filling it in memory much the same way as with the DOM-like

functionality but periodically pushing completed parts to disk and removing them from memory.

With these techniques the importing and exporting API's use the same objects regardless of the size of the dataset, making it easier to evolve a program to handle larger XML documents. These techniques have been implemented in a Ruby framework released as GPL-licensed free software[1,8,10].

2 Mapping XML to Objects

To map a XML format to an object structure a simple approach has been chosen. Each element in the format corresponds to a class and each attribute or sub-element is reached by an accessor in the class. Attributes can be modelled as simple get/set style methods. The sub-elements are stored in a list in the parent element, so as to support the preservation of order. To be able to handle mixed-content elements this list may also contain strings.

No attempt at type conversion was made, so everything is either text or an object representing an element. This is not however a limitation of the technique but of the implementation. We expect that existing approaches in this area[12] would integrate easily with our work.

The implementation is fully functional. Here is an example of mapping a simple XML format into Ruby Objects:

```
require 'rubygems'
require 'xmlcodec'

class SimpleFormat < XMLCodec::XMLElement
  xmlformat 'Race Scores'
end

class Race < SimpleFormat
  elname 'race'
  xmlattr :name
  xmlsubelements
end

class Car < SimpleFormat
  elname 'car'
  xmlattr :name
  xmlattr :number
  xmlsubel :result
end

class Result < SimpleFormat
  elname 'result'
  xmlattr :time
  xmlattr :place
end
```

Although this might look like pseudo-code it is actually fully functional Ruby code. The `elname`, `xmlformat`, `xmlattr`, `xmlsubel` and `xmlsubelements` lines are calls to class level functions inherited from `XMLElement` that are executed when the classes are interpreted and setup all of the XML import/export functionality. If we wanted to create some race results with two cars we'd write:

```
race = Race.new

car1 = Car.new
car1.name = 'Speeding Bullet'
car1.number = 17
result1 = car1.result = Result.new
result1.time = '1 hour 12 min'
result1.place = 2
race << car1

car2 = Car.new
car2.name = 'Overspeeding Bullet'
car2.number = 12
result2 = car2.result = Result.new
result2.time = '1 hour 11 min'
result2.place = 1
race << car2
```

From now on the `race` object will be completely filled with the race information. To generate XML all that's needed is to run `race.xml_text`, which will generate the following document:

```
<?xml version="1.0"?>
<race>
  <car number="17" name="Speeding Bullet">
    <result time="1 hour 12 min" place="2"/>
  </car>
  <car number="12" name="Overspeeding Bullet">
    <result time="1 hour 11 min" place="1"/>
  </car>
</race>
```

To import a XML document something simple like this can be done:

```
xmltext = '<race></race>'
race = SimpleFormat.import_xml_text(xmltext)
```

The `race` object will link to the full representation of the XML tree. The framework includes a little more functionality than was explained here including importing/exporting from a DOM object model instead of XML text. See the framework website and the API docs for more information[10,11].

3 Event Parsing Using Objects

In the previous section we explained how the mapping between XML and objects has been solved. Now we take those same objects and use them together with a standard XML stream parser to produce an event-based XML parser where each event returns a complete object.

The parser is implemented by keeping the current XML element as well as all its ancestors in a stack. The top of the stack will always contain the element currently being parsed. Whenever a new XML element starts a new object is created in the stack and it is connected to its parent so that the XML tree is recreated in memory. Any text content is also added.

Every time a XML element closes we have a complete representation of it in memory and can thus generate an event ourselves to any listener, informing that a full element has been parsed and passing it along to be used.

If nothing else is done the event that closes the root element of the XML document returns an object that indirectly links the whole tree. Although this might be useful for some purposes it would impose a memory limit on the size of the document that could be parsed. To solve this a listener can choose to consume the element during the processing of the event. In this case what happens is that the element is removed from the tree and will not appear as one of the children of its parent when the event for that element comes.

This technique will allow the parsing of an unlimited sized document with memory usage proportional to the maximum depth of the XML tree. If the tree has unbounded depth the memory usage will not be bounded and we won't be able to parse unlimited sized documents. In the real-world big XML documents tend to be very wide instead of deep.

If you do find such a difficult case of an XML document that is huge by being deeply nested even a simple stream parser will consume unbounded amounts of memory and a totally different parsing technique needs to be employed or perhaps discovered.

4 Generating Unlimited-Sized Documents with Objects

Now that we've seen how it is possible to parse a document of unlimited size with low memory usage we need a way to generate such a document. There is no usual equivalent to the stream parser for document generation. Software tends to use DOM when feasible and its own exporter when necessary.

The XML to object mapping we've described can be used the same way as DOM is usually used with the benefit of a better API. To do this we first create the whole XML tree in memory, represented by objects, and then write it to disk in one go.

To be able to use the same API and not keep the whole document in memory while we create it we'd have to implement a fairly complicated virtual memory system so that we could transfer elements back and forth from disk. To make this much simpler all we have to do is impose the restriction that the tree is

filled depth-first and left-to-right. This is the order used in the XML document itself.

If the in memory tree is filled in the same order as the elements appear in the XML text we can continuously write elements to disk and remove them from memory. To do this we use the API as usual but call for the partial export of an element when we are done with it.

What the partial export does is append the text of the element to the XML file and then remove the element from its parent. To be able to do this though we must first at least write the opening tags of the parent elements. This is handled by separating the partial export process into two steps. The first step writes the opening tag for the element with any attributes followed by any sub-elements that have been added to the parent. The second step writes the close tag for the element and removes it from the parent. Between these two steps more elements can be added to the element and exported to disk.

We don't have to fill the tree in the exact same order as the XML text is written. The actual restriction is that after we call to export a certain element we must not change or add any element that would be before it in the XML text since that point in the file has already passed.

Once again this allows the creation of unlimited sized documents with memory usage proportional to the depth of the tree

5 Practical Application

This framework and techniques have been created to solve the problem of creating and interpreting large EAD[13] XML files. Both scalable import and export techniques were used to produce and consume EAD files with sizes of over 100 megabytes.

Because the framework has been extracted from the concrete solution the separation between the two proved very clean. Import/export techniques were first implemented for the EAD case and later made generic. The result is that in the end the EAD library became just a set of class definitions for the format's elements, like the simple example given to illustrate the XML to object mapping. This library has also been released as free software[9].

This application shows that an import/export library can be defined for a specific XML format without hand-coding a lot of infrastructure and still get a scalable library, capable of handling size-unlimited XML files with no size limits.

The library has been developed to handle the EAD creation and parsing needs at *Instituto dos Arquivos Nacionais/Torre do Tombo* (IANTT). The production archive system uses a proprietary import/export format. The two main goals for EAD use where to export and convert the full set of existing descriptions into the format and then make them available in a Web application with full search capabilities.

We'll now see how the import and export API's can be used in a real use scenario to import and export large quantities of EAD. No deep knowledge of the EAD format is required to understand these examples. It suffices for this purpose

to understand that EAD is an XML format to describe a tree of records. Each record is an `archdesc` XML element if it's the root or `c` element otherwise. The tree is described in XML by nesting the `c` elements.

5.1 Generating Large Quantities of EAD XML Files

The archive system at IANTT has an underlying data model based entirely on text fields. Each record is just a set of strings for the fields. Connections between fields are done by brute-force or indexed search over these fields. The record tree is inferred from the reference codes rather than being kept by linking records to each other.

Exporting the contents of the archive system results in a single file containing all the records in sequence. This file has been indexed so that it could be traversed in the lexicographic order of the reference codes for each record. This is the same as visiting the tree of records in pre-order because reference codes are defined much like Unix paths. For example `PT-TT-TSO/IC` is the immediate child of `PT-TT-TSO`.

Using the index of the records the method described in Section 4 was used. For each of the roots of the description tree a single EAD file containing the corresponding tree has been created as follows:

```
1  @curdepth = -1
2  @index.each_with_depth do |obj, depth|
3    if depth == 0
4      change_file(obj)
5    else
6      newel = EADCodec::Level.new
7      if depth > @curdepth
8        @curelements.push(newel)
9      else # depth <= curdepth
10       (@curdepth - depth + 1).times do
11         @curelements[-1].end_partial_export(@file)
12         @curelements.pop
13       end
14       @curelements[-1] << newel
15       @curelements.push(newel)
16     end
17   end
18   @curdepth = depth
19
20   fill_element(@curelements[-1], obj)
21 end
```

Line 2 shows an iteration over every single record. This iteration receives for each element an object encapsulating the text fields of the record and the tree depth we are currently in. The depth is calculated using the reference code. For each record a new object is created representing the EAD element (line 6).

All the ancestors of the element currently being processed are kept in a stack. To maintain this stack we first find out if the depth of the tree has increased (line 7). If so the new element is pushed into the stack. If not the stack is popped to the depth immediately above the one we're currently at and each of the elements is finished and written to disk (lines 10 through 13). The current element is then added to the EAD tree (line 14) and pushed into the stack (line 15).

At the end of each iteration the depth is saved to be used on the next (line 18). Finally the EAD XML element is filled with the contents from the record (line 20). This implements the mapping between the archive system's set of fields and the EAD format. `fill_element` is a rather large method so it will not be listed here. It's operation is however quite simple. The partial export API (as all the API's described) uses the same objects as the more traditional API's so the method fills the object in much the same way as was done in Section 2.

The tree depth is zero when we're in one of the roots so we change the file that's currently being written (lines 3 and 4). `change_file` is defined as:

```
1  def change_file(obj)
2    @ead.end_partial_export(@file) if @ead
3    @file = File.new(obj.RefNo+'.xml', 'w')
4    @ead = EADCodec::Document.new(obj.RefNo, '...')
5    @curelements = [@ead.archdesc]
6  end
```

`change_file` starts by finishing the export of the previous file by calling for the end of the export on the root element (line 2). The framework takes care of recursing down the XML tree and finishing all of its still open elements. After that a new file is created in the filesystem (line 3) and a new EAD document object is created (line 4). The element stack is initialised with a single element, the root of the element tree which is the `archdesc` element (line 5). At this point we could do a call to the partial export methods to export the beginning of the EAD file. This is actually unnecessary since it will be taken care of automatically by the framework the first time an element is exported (line 11 of the previous code listing).

The export from the archive system is about 800 megabytes. After conversion to EAD through the process just described a few hundred files are created with a total size of around 300 megabytes. The process takes approximately 2 hours on a modest computer (3.0Ghz Celeron processor with 1 GiB of memory), most of which is spent indexing the exported file and not doing the conversion itself.

The conversion process takes time proportional to the number of records in time and constant space. The indexing process makes the full conversion process have memory usage linearly proportional to the number of records because it needs to create an index of all of them, sorted by reference code. Time is of NlogN order because of the sorting step. This turned out not to be a problem. The index is small; all it stores is the records' codes along with their offset within the file so that their contents can be retrieved.

Most of the complexity of this process lies in recreating the tree from the flat set of records the archive system provides us with. Besides the `fill_element` method there's just three lines of code that deal with the EAD exporting (line 11 in the export loop and lines 2 and 4 in `change_file`). `fill_element` itself doesn't do any calls to the partial export process and could be used without modification with the more classic export API's. The partial export API is thus very simple to use although it does require care in setting up the correct looping procedure to serially generate the XML tree.

5.2 Parsing and Importing Large Quantities of EAD XML Files

After generating these EAD files a Web application has been created to index and search them. We'll now see how large EAD files can be parsed and processed using the event parser described in Section 3. The code to implement this is extremely simple:

```
1  files.each do |file|
2    l = MyEADStreamListener.new
3    el = EADElement
4    parser = XMLStreamObjectParser.new(el, l)
5    parser.parse(File.new(file))
6  end
7
8  class MyEADStreamListener
9    def el_c(el)
10     handle_object(el)
11   end
12
13   def el_archdesc(el)
14     handle_object(el)
15   end
16
17   def handle_object(o)
18     # ... Do something with this description ... #
19     o.consume
20   end
21 end
```

The main loop (lines 1 through 5) is very simple. For each of the files it creates a listener object (line 2), then a stream parser with the listener and the root element for the EAD format (line 4). Finally the parser is told to parse the file by passing it a file object (line 5).

The stream parser will parse each of the XML files and for each element within them call the listener. The API works almost the same way as a SAX parser. The difference is that the events are full elements instead of the opening and closing of tags. The listener will contain one or several methods named

`el_tagname`. For each of the elements found in the XML file the parser will call the corresponding method in the listener if it exists.

Lines 8 through 21 show an example of a listener for the EAD format. There are methods to listen to `c` and `archdesc` elements that both call `handle_object` to do the actual processing since these elements are similar. `handle_object` will do something with the object and then consume it (line 19) removing it from the XML tree and allowing the memory for it to be freed.

In the Web application `handle_object` adds a record to a database containing the contents of the description as well as adding it to a textual index for searching. Full imports of large EAD files have been successfully completed. Memory usage is constant and running time is proportional to the number of records in the EAD file.

The final version of the application ended up not using this parser. Because of unrelated architectural decisions it sufficed to use a simpler parser of the same type that instead of returning elements as Ruby objects containing the parsed XML content just returned the XML text itself. This could be achieved by doing `o.xml_text` inside `handle_object`. It is however much more efficient to not create the objects to represent the XML structure.

6 Comparison with Other Approaches

We will now compare the existing technologies to the proposed solution. Table 1 shows six different technologies and a set of comparison points. The first two columns indicate whether the API's are capable of both parsing and generating XML. All but SAX do both. The Generating equivalent of SAX is usually to output XML manually by writing the file directly. StAX has a much friendlier way to do essentially the same thing while DOM does it by using the same in-memory structure it creates when parsing to generate XML.

Name	*Parse*	*Generate*	*Mapping*	*Validating*	*API Type*	*Space Used*
XMLCodec	**Yes**	**Yes**	**Yes**	No	Push	**O(Depth)**
JAXB	**Yes**	**Yes**	**Yes**	**Yes**	Push	O(Size)
XML Beans	**Yes**	**Yes**	**Yes**	**Yes**	Push	O(Size)
StAX	**Yes**	**Yes**	No	No	Push	**O(1)**
DOM	**Yes**	**Yes**	No	No	Push	O(Size)
SAX	**Yes**	No	No	No	**Pull**	**O(1)**

Table 1. Feature Matrix of the Various Approaches

JAXB, XML Beans and XMLCodec are all capable of performing the mapping between XML and objects. In this respect our proposal has the most modest feature set. JAXB and XML Beans support full type systems and richer API's. Somewhat as a result of this they also support validating the XML while our solution does not.

StAX, unlike all others, has a push-style API. This means that the control of the advance of the XML parsing process is done by the caller and not the API.

As for the space usage or complexity, JAXB, XML Beans and DOM all use space proportional to the XML document size, since they load it all into memory at once. As we've shown our approach has space complexity proportional to the document's depth. StAX and SAX use constant space since they don't need to keep the state of the XML tree around. This is not usually an actual advantage because most of the applications that use these API's will almost surely have to add a layer above them that will at least keep track of all the elements in the current depth expansion of the XML tree and will thus use space proportional to it's depth.

Our solution equals or improves the common ones over most of the criteria with which XML processing API's are usually compared. It is behind in some areas not because of inherent problems with the approach but because the current implementation is still not full-featured. The only area where the approach has an actual inherent limitation is in space efficiency. As we've seen this is only important in a very small number of cases where SAX or StAX low-level parsers will have to be used. When compared to its most natural competitors like JAXB or XML Beans its space efficiency is a clear improvement.

7 Further Work

The XML mapping to objects and the import/export API's were built out of necessity to solve a particular problem set. The implementation turned out stable and functional enough to suggest several avenues of further work.

7.1 Performance Work

All of the work on performance improvement centred around optimising for space and not time. The technique itself is not algorithmically complex but the current implementation is somewhat naive and unoptimised. There has been some work done in caching some frequent operations and some simple code optimisations. Profiling and optimisation work could probably speed it up a fair amount.

7.2 Validation

The XML mapping has no support for the implementation of validation of XML elements. It would be simple to add support for checks performed when importing and exporting elements.

7.3 Pull-Style API

StAX is different from all other common XML API's in that it gives control over the advance of the parsing process to the caller. This is orthogonal to the parsing

technique described. Our implementation uses a SAX parser but the technique can be just as easily implemented using StAX.

The API for the pull parser would reverse the calling process. Instead of the caller providing a listener to respond to parser events it would call the parser repeatedly to process the XML. Each call would return an object representing a XML element instead of a XML instruction, tag or text. This new parser would be to StAX what our parser is to SAX.

7.4 Auto-Generation from Schemas or DTDs

Writing a XML mapping requires writing a class for each of the elements in the format. It would be feasible to automatically create these classes from a XML schema or DTD. It would then be possible to create the validation rules automatically. This would be similar to what XMLBeans and JAXB do [5,7].

7.5 Generic XML Import/Export

Although the framework has been generalized to work with any XML format it still requires classes to be defined for each element type, instantiating the framework as an import/export library for a specific format. This isn't strictly required for the mapping techniques to work. It would be possible, and relatively easy, to create a generic element type that is instantiated with the element name and to which attributes and elements are added. The import and export techniques could still work with it.

Supporting generic XML import/export would turn the framework into a DOM-style API with the possibility of using the techniques described for handling large documents.

7.6 Virtual-Memory Style Import and Export

As briefly pointed out when describing the framework, the partial export method avoids having to implement a full VM-style abstraction by limiting the filling of the in memory structure to the XML text order and by removing elements from memory after they've been exported to disk. Importing a large document is handled by a stream parser whose events are complete elements. This is useful in a large number of situations but might require several passes through a document for some workloads.

A unified solution to the limitations imposed by these methods would be to treat the XML document as the on-disk representation of the in memory structure and implement the VM-style abstraction that would make that distinction transparent to the API.

This is feasible if not trivial but much care would be needed with the space-time trade-off of such an implementation. The techniques presented were designed explicitly to be both space and time efficient by limiting the XML processing to use the text order. Random access requires navigating back and forth in the structure.

If we keep a full skeleton structure in memory, random access will be fast and memory-hungry since it will only hit the disk to fetch content. Another choice is to only keep in memory whatever element pointers the user has. This will be memory-lean but slow because it must hit disk for every navigation within the tree. A compromise between the two can probably be made by going with the second option but introducing a dynamically sized cache that can be configured to a desired size.

8 Conclusion

The techniques explained aren't just proposals, a fully-functional implementation exists and they've been tested in production environments with large documents. It has been shown that mapping XML to objects can be simple yet API-rich. The techniques for importing and exporting large XML documents have proved useful and shown adequate performance. Further work along these lines could completely break the current separation between DOM and SAX techniques and how they are used.

Although the techniques described are independent of the programming language the current implementation and the way that the XML mapping works would not be possible without the expressiveness of Ruby. All of what's described here has been implemented in less than a thousand lines of code and has been fully unit tested in less than 800 lines. It has also been a joy to write.

References

1. Ruby Programming Language. `http://www.ruby-lang.org/en/`.
2. Simple API for XML. `http://www.saxproject.org/`.
3. W3C Document Object Model. `http://www.w3.org/DOM/`.
4. XML Schema. `http://www.w3.org/XML/Schema`.
5. XMLBeans. `http://xmlbeans.apache.org/overview.html`.
6. BEA Systems Inc. JSR 173: Streaming API for XML. `http://jcp.org/en/jsr/detail?id=173`, October 2003.
7. Ed Ort and Bhakti Mehta. Java Architecture for XML Binding (JAXB). `http://java.sun.com/developer/technicalArticles/WebServices/jaxb/`, March 2003.
8. Free Software Foundation. GNU General Public License. `http://www.gnu.org/copyleft/gpl.html`, June 1991.
9. Pedro Côrte-Real. eadcodec - an EAD importer/exporter library for Ruby. `http://eadcodec.rubyforge.org/`.
10. Pedro Côrte-Real. xmlcodec - a XML importer/exporter framework for Ruby. `http://xmlcodec.rubyforge.org/`.
11. Pedro Côrte-Real. xmlcodec API. `http://xmlcodec.rubyforge.org/doc/`.
12. R. Connor, D. Lievens, P. Manghi, and F. Simeoni. Extracting typed values from XML data, September 2001.
13. SAA Encoded Archival Description Working Group. *Encoded Archival Description Tag Library.* The Society of American Archivists, August 2002.
14. F. Simeoni, D. Lievens, R. Connor, and P. Manghi. Language Bindings to XML, 2003.