Algorithm for XML Compression using DTD and Stack

G. M. Tere
Department of Computer Science,
Shivaji University, Kolhapur,
Maharashtra – 416004, India

B. T. Jadhav
Department of Electronics and Computer Science,
Y.C. Institute of Science,
Satara, Maharashtra – 415001, India

ABSTRACT
Worldwide standard for data definition is XML. For developing SOA based applications XML is extensively used. SOA based applications contains many different applications which are integrated to each other. For solving the problem of interoperability XML documents are used. XML is widely used for a variety of tasks, including configuration files, protocols, and web services. XML has problem with processing. It is verbose nature. Simple messages can be quite large, containing very small information. In XML documents lots of information are duplicated, which take more computing resources and thus performance of web services decreases. Lots of research is going on regarding how to process XML, so that web services' performance can increase. We present an algorithm for compressing XML documents using Document Type Definition (DTD) specifications. Our algorithm is based on lossless compression technique. The model used for compression and decompression is generated automatically from the DTD, and is used in conjunction with an arithmetic encoder to produce a compressed XML document. Our compression technique is on-line, that is, it can compress the document as it is being read. We have implemented the compressor generator, and we have mentioned the results of our experiments performed with XML documents created from Oracle database. The average compression is better than that of XMLPPM and XMill. The processor, XPrFAST, is able to compress large documents where XMLPPM failed to work as it ran out of memory. The technique we have proposed is simple and effective and we have compared it with XMLPPM and XMill.

General Terms
Software Engineering, XML Processing, SOA

Keywords
Arithmetic coding, compression ratio, DTD, DFA, XMLPrFAST.

1. INTRODUCTION
XML has been standardized by W3C. It is world wide standard for data definition and description. Thus whenever there is need of data exchange between different applications developed using different platforms. XML is widely used in developing web services which are required for loose coupling of different applications. Considering the importance of XML there is need to process XML documents efficiently. Main problem of XML is that of verbose nature. It is easy for human to understand the XML documents as they are plain text files, but it is challenging for computer to process them. We need to present necessary information using less data. Large document size means that the amount of information that has to be transmitted, processed, stored, and queried is often larger than that of other data formats [1]. If to present same information more data is required then we neeed to compress the data, so that communication between applications will not demand more bandwidth. XML document's structure is specified by DTD. The purpose of DTD (Document Type Definition) is to define the legal building blocks of an XML document. We have studied different models for the compression of XML documents.

As data in business applications is growing, we need to exchange and process large XML documents, and therefore theres is need of efficiently compressing XML documents. The syntax directed translation scheme converts the DTD into a set of Deterministic Finite Automata (DFA) one for each element in the DTD [2]. Each transition is labeled by an element, and the action associated with a transition is a call to a simulator for the DFA for the element labeling that transition. Every element that has some attributes or character data has an associated container. The scheme we describe automatically groups all data for the same element into a single container which is compressed incrementally using a single model for compressor[20]. We have performed experiments with XML documents containing data from Oracle databases [13]. We then compared performance of our tool with that of two well known XML-aware compression schemes, XMill[12] and XMLPPM [6]. For experiments we have used OE schema of Oracle 11g [13] and DBLP [7] and UniProt [20] database. The XML documents are stored in the Oracle XML DB Repository after validation against the registered XML schema purchaseorder.xsd. The purchase order XML documents are located in the Oracle XML DB Repository folder $ORACLE_HOME/rdbms/demo/order_entry/2002/month, where month is Jan, Feb, Mar, and so on. For dealing with XML data in Oracle we have to use SYS.XMLTYPE.The tool XMLPPM could not compress UniProt database as it ran out of memory. The average compression ratio of our scheme is better than that of XMLPPM and significantly better than that of XMill. There is no best tool available for compressing XML documents. Our tool took more time and memory sd compared with XMill. This is because of drawback of a scheme based on arithmetic coding, which has to perform lengthy table updating operations after reading every symbol. However XMill cannot perform on-line compression as can XMLPPM and our tool XPrFAST (XML Processor with Finite Automata and Stack). Online compression is useful for

processing large XML documents. Section 2 describes related work. Section 3 describes arithmetic coding. In section 4 the structure of XML documents and DTD is discussed. In section 5 we have presented and analyzed the experimental results obtained with different compression tools like ith those of XMill and XMLPPM and that of a general purpose compressor bzip2[2].

2. RELATED WORK

Data and information are not synonymous terms! Data is the means by which information is conveyed. Data compression aims to reduce the amount of data required to represent a given quantity of information while preserving as much information as possible. Cameron has used Context Free Grammars, CFG, for compressing files [4]. Given estimates for derivation step probabilities, he has shown how to construct practical encoding systems for compression of programs whose syntax is defined by a CFG. The models are, however, fairly complex in their operation. For the scheme to be effective, these probabilities have to be learned on sample text. Syntax based schemes have also been used for machine code compression [16][17][18]. With a DTD, each of XML files can carry a description of its own format. With a DTD, independent groups of people can agree to use a standard DTD for interchanging data. Application can use a standard DTD to verify that the data received from the outside world is valid. The XML-specific compression schemes that we are aware of are XMLZIP[24], Xmill and XMLPPM. The last two have tried to take advantage of the structure in XML data by either transforming the file after parsing, breaking up the tree into components [12] or injecting hierarchical element structure symbols into a model that multiplexes several models based on the syntactic structure of XML [6]. They do not require the DTD to compress the document, and even if it is available it is not used. XMLZIP parses XML data and creates the underlying tree. It then breaks up the tree into many components, the root component at depth d and a component for each of the sub trees at depth d.

3. ARITHMETIC CODING AND FINITE AUTOMATA

3.1 Arithmetic Coding

Arithmetic coding does not replace every input symbol with a specific code [15]. Instead it processes a stream of input symbols and replaces it with a single floating point output number. The longer (and more complex) the message, the more bits are needed in the output number. The output from an arithmetic coding process is a single number less than 1 and greater than or equal to 0. This single number can be uniquely decoded to create the exact stream of symbols that went into its construction. In order to construct the output number, the symbols being encoded need to have a set of probabilities assigned to them. Initially the range of the message is the interval [0, 1). As each symbol is processed, the range is narrowed to that portion of it allocated to the symbol. As the number of symbols in the message increases, the interval used to represent it becomes smaller. Smaller intervals require more information units (i.e., bits) to be represented.

3.2 Finite Automata

A deterministic finite state automaton (DFA) is a simple language recognition device. It can be seen as a machine working to give an indication about strings which are given in input or it can be given a mathematical definition. Strings are fed into the device by means of an input tape, which is divided into squares, each one holding one symbol. The main part of the machine itself is a black box which is, at any specified moment, in one of a finite number of distinct internal states, among which we distinguish an initial state and some final states. This black box, called the finite control, can sense what symbol is written at any position of the input tape by means of a movable reading head. Initially, the reading head is placed at the leftmost square of the tape and the finite control is set in a designated initial state.

In a finite context scheme, the probabilities of each symbol are calculated based on the context the symbol appears in. In its traditional setting, the context is just the symbols that have been previously encountered. The order of the model refers to the number of previous symbols that make up the context. In an adaptive order k model, both the compressor and the decompresser start with the same model. The compressor encodes a symbol using the existing model and then updates the model to account for the new symbol. Typically a model is a set of frequency tables one for each context. After seeing a symbol the frequency counts in the tables are updated. The frequency counts are used to approximate the probabilities and the scheme is adaptive because this is being done as the symbols are being scanned. The decompresser similarly decodes a symbol using the existing model and then updates the model. Since there are potentially qk possibilities for level k contexts where q is the size of the symbol space, update can be a costly process, and the tables consume a large amount of space. This causes arithmetic coding to be somewhat slower than dictionary based schemes like the Ziv-Lempel[24] scheme.

4. REPRESENTATION OF XML DOCUMENTS USING FINITE AUTOMATA

XML documents contain element tags which include start tags like <name> and end tags like </name>. Elements can nest other elements and therefore a tree structure can be associated with an XML document. Elements can also contain plain text, comments and special processing instructions for XML processors. In addition, opening element tags can have attributes with values such as gender in <person gender=‘‘male’’>. Detailed specifications are given in [23]. XML documents have to conform to a specified syntax usually in the form of a DTD. Usually XML documents are parsed to ensure that only valid data reaches an application. Most XML parsing libraries use either the SAX interface 286 or the DOM (Document Object Model) interface. SAX is an event based interface suitable for search tools and algorithms that need one pass. **SAX parser** is work differently with DOM parser, it either load any XML document into memory or create any object representation of the XML document. Instead, the SAX parser use callback function.

The DOM model on the other hand is suitable for algorithms that have to make multiple passes. Since XML documents are stored as plain text files one possibility is to use standard compression tools like bzip2. Cheney[6] has performed a study of the compression using such general purpose tools and observed that each general purpose compressor performs poorly on at least one document. Since XML documents are governed by a rather restrictive set of rules the obvious way to go, is to try to use the rules to predict what symbols to expect. Further if the rules are already known a-priori then the compressor which is tuned to take advantage of the rules can be generated directly from the rules themselves. This is what we achieve in our scheme XPrFAST. The scheme proposed in this paper assumes that the DTD describing the data is known to both the sender and the receiver. Typically, an element of a DTD consists of distinct beginning and ending tags enclosing regular expressions over other elements. Elements can also contain plain text, comments and special instructions for XML processors. Opening element tags can have attributes with values.

Example 1. Consider a DTD defined as follows:

```
<!DOCTYPE Diary[
<!ELEMENT Diary (person*)>
<!ELEMENT person ((name | (firstName,
                lastName)),
                email, contactno, desig?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT firstName (#PCDATA)>
<!ELEMENT lastName (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT contactno (#PCDATA)>
<!ELEMENT desig (#PCDATA)>
]>
```

PCDATA means parsed character data. Think of character data as the text found between the start tag and the end tag of an XML element. PCDATA is text that WILL be parsed by a parser. The text will be examined by the parser for entities and markup. Tags inside the text will be treated as markup and entities will be expanded. However, parsed character data should not contain any &, <, or > characters; these need to be represented by the & < and > entities, respectively. Below is an instance of an XML document conforming to this DTD.

```
<Diary>
<person>
<firstName>Neeta</firstName>
<lastName>Singh</lastName>
<email>neeta_singh@yahoo.co.in</email>
</person>
<person>
<name>Milind Joshi</name>
<email>milind.joshi@gmail.com</email>
<desig>Programmer</desig>
<contactno>09930335566</contactno>
</person>
</Diary>
```

The strings following each element declaration are just regular expressions over element names and therefore each of them can be associated with a DFA.

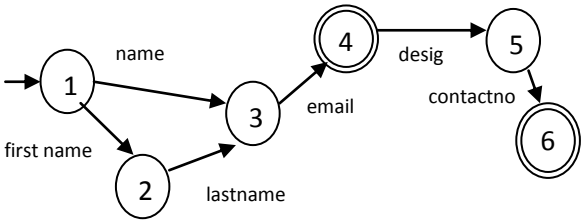

Fig. 1 DFA for the right hand side of the production for nn in Example 1

The DFA for the right hand side of the rule for element card is shown in Fig. 1. There are two kinds of states in this automaton, those having a single output transition and those with multiple output transitions. Symbols that label single output transitions need not be encoded as their probability is 1. Thus encoding of symbols by the arithmetic compressor needs to be performed only at states with more than one outgoing transition. An arithmetic encoding procedure is called at each such state for each element. As we observed in Section 3, the arithmetic encoder maintains tables of frequencies which it updates each time it encodes a symbol. Each element which has a #PCDATA attribute will result in a call to an arithmetic encoder which uses a common model for all instances of that element attribute and encodes them using the same set of frequency tables. A typical sequence of actions is then as follows: Enter the start state of a DFA representing the right side of a rule; if there is only one edge out of the state then do nothing; if that element has a #PCDATA attribute then encode the string of symbols using the frequency tables associated with that element; if there is more than one edge encode the element labeling the edge taken, using an arithmetic encoder for that state, and transit to the the start state of the DFA for that element; the decoder mimics the action of the encoder generating symbols that are certain and using the arithmetic decoder for symbols that are not. XPrFAST uses a single container for the character data associated with each element though this has the capability to use the context (i.e. the path along which it reached this element). The reason is best illustrated by the example below:

Example 2. Consider the element below

```
<!ELEMENT Project (date, date, ...)>
<!ELEMENT Employee (date, ...)>
<!ELEMENT date (#PCDATA)>
```

The date in Employee is the joining date. The first and second date in Project is the starting and ending dates respectively of the project. XPrFAST uses a single model for date and the reason is clear. Experimentation indicates that having different models for date in this case is counter-productive as different models for essentially the same kind of data consume an inordinate amount of memory with little or no gain in compression ratio.

4.1 Compression and Decompression Using XPrFAST

A state of the compressor is a pair (element, state) where element represents the current element whose DFA XPrFAST is traversing and state of the DFA where it currently is. We have used the work done by Hariharan and Priti [25], but we have done experiments with different data. Assume that the current state of the Encoder is (i, j). When an open tag is encountered for element k in the document, the current state pair of the encoder is stored on the calling stack and the DFA for the element k is entered. The current state of the encoder now becomes (k, 0). When the end tag is encountered for element k, the stack is popped and the new state of the encoder becomes (i, j + 1). As mentioned earlier, tags are not encoded if the number of output transitions is equal to 1. For example, for the case below we need not encode the tag D but we have to encode B and C.

<!ELEMENT A ((B | C), D)>

Every state has an arithmetic model which it uses to encode the next state. Note that this is different from the model used to encode character data, which is handled as described below.

Consider the element below.

<!ELEMENT A ((#PCDATA A|B)*)>

There are two transitions from the start state of the DFA for element A. One of them invokes the arithmetic model for CDATA which is common for all PCDATA associated with any instance of element A in the document. The other transition invokes the DFA for element B after pushing the current state in the stack. We have used the algorithm developed by Hariharan and Priti [25] for designing our compression tool XPrFAST.

```
void Encoder(){
ExitLoop = false;
//StateStruct is a pair of int(ElementIndex, StateIndex)
//ElementIndex represents the automaton
//StateIndex is the state in the above automaton
StateStruct CurrState(0, 0);
while(ExitLoop == false)
{
     Type = GetNextType(FilePointer, ElementIndex);
     switch(Type)
    {
        case OPENTAG:
          //Encode ElementIndex in CurrState context
          EncodeOpenTag(CurrState, ElementIndex);
          Stack.push(CurrState);
          CurrState = StateStruct(ElementIndex, 0);
          break;

       case CLOSETAG:
       //Encode CLOSETAG in CurrState context
       EncodeCloseTag(CurrState);
       if(Stack.empty() == true)
      {
           ExitLoop = true;
       }
       else
       {
           CurrState = Stack.pop();
           //Make state transition in CurrState.ElementIndex
           //automaton and get the next state
           CurrState.StateIndex =
                 MakeStateTransition(CurrState,
           ElementIndex);
       }
       break;

       case PCDATA:
       //Encode Pcdata in Currstate context
       EncodePcdata(CurrState);
       CurrState.StateIndex =
            MakeStateTransition(CurrState, PCDATA);
       break;
    }
  }
}
```

Fig. 2 Algorithm for compressing XML documents [25]

```
void Decoder()
{
ExitLoop = false;
StateStruct CurrState(0, 0);
while(ExitLoop == false)
{
     //Decode the type in CurrState context
     Type = DecodeNextType(FilePointer, CurrState,
              ElementIndex);
     switch(Type)
     {
          case OPENTAG:
            //Write open tag of the Element of ElementIndex
            WriteOpenTag(ElementIndex);
            Stack.push(CurrState);
          CurrState = StateStruct(ElementIndex, 0);
          break;

       case CLOSETAG:
       //Write close tag of the Element of ElementIndex
          WriteCloseTag(ElementIndex);
          if(Stack.empty() == true)
          {
               ExitLoop = true;
          }
          else
          {
               CurrState = Stack.pop();
               CurrState.StateIndex =
                     MakeStateTransition(CurrState,
               ElementIndex);
          }
          break;

          case PCDATA:
               DecodePcdata(CurrState);
               CurrState.StateIndex =
               MakeStateTransition(CurrState, PCDATA);
          break;
       }
    }
}
```

Fig. 3 Algorithm for decompressing XML documents [25]

5. EXPERIMENTAL RESULTS

We have examined the performance of three tools XMill, XMLPPM and XPrFAST on five large XML documents. The experiments were done on DELL laptop Core2 Duo, Intel Pentium IV 2 GHz with 4 GB RAM , Windows XP was OS. The sizes of these documents are displayed in Table 1. We define the Compression Ratio as the ratio of the size of the compressed document to the size of the original document

expressed as a percentage. The compression ratios for all three schemes are shown in Fig. 3 along with that of a general purpose compressor bzip2. The compression ratios of XPrFAST and XMLPPM are considerably better than that of XMill for all but one of the documents. XMLPPM, however, ran out of memory for two documents. It also takes significantly longer than XPrFAST whereas XMill is more efficient in terms of space and time. The disadvantage of XMill is that it cannot perform on-line compression. Therefore it is not suitable for compressing large XML data. We expect that our scheme will do well wherever the markup content is high as tags whose probability of occurrence is 1 are not included in the compressed stream. Fig. 3 also shows the compression ratios for tags alone. XPrFAST compresses tags more efficiently than in other schemes. Time required in sec for compressing different XML documents by different XML compression tools is shoen in Table III and in Fig. 4.

TABLE I
SIZES OF XML DOCUMENTS THAT WERE COMPRESSED

Name	Size in MB
OE	542
Dblp	253
Uniprot	1070

TABLE II
COMPRESSION RATIO OF DIFFERENT XML DOCUMENTS BY DIFFERENT COMPRESSING TOOLS

	Compression ratio for different XML documents			
XML doc	XPrfAST	XMLPPM	Xmill	bzip2
OE	18.50%	16.53%	30.45 %	23.72 %
dblp	10.00%	10.00%	14.80 %	11.60 %
uniprot	7.50%		8.00%	8.80%

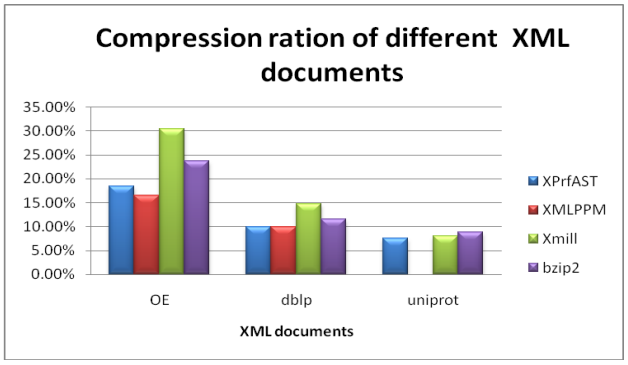

Fig. 3. Compression ratio of different XML documents for different compressing tools

TABLE III
TIME IN SEC REQUIRED BY DIFFERENT COMPRESSION TOOLS FOR COMPRESSING DIFFERENT XML DOCUMENTS

	Time measured by different Compression tools in sec		
XML doc	XPrfAST	XMLPPM	Xmill
OE	206	513	33
Dblp	324	1766	45
uniprot	1252		112

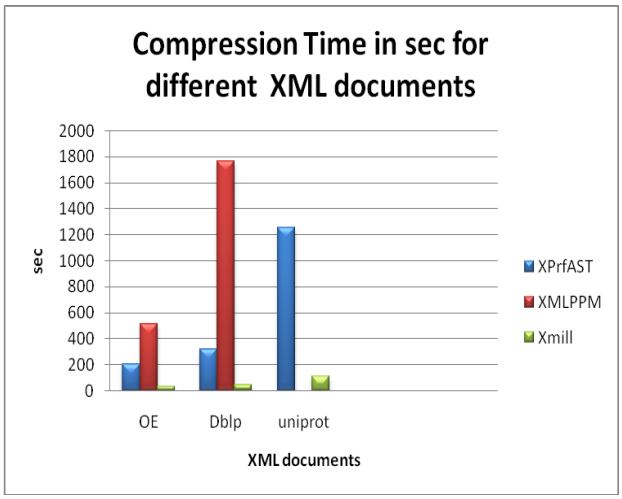

Fig. 4. Time in sec required by different compression tools for compressing different XML documents

XPrFAST does not need a SAX parser as do XMill and XMLPPM as some form of parsing is already embedded in its action. XMLPPM ran out of memory for uniprot.xml and mich.xml. Running times are shown for only XML-aware schemes.

We have presented a scheme for the compression of XML documents where the underlying arithmetic model for the compression of tags is a finite state automaton generated directly from the DTD of the document. The model is automatically switched on transiting from one automaton to another storing enough information on the stack so that return to the right state is possible; this ensures that the correct model is always used for compression. On return, the stack is used to recover the state from which a transition was made. Our technique directly generates the compressor fromthe DTD in the appropriate format with no user interaction except the input of the DTD. Our experiments on different databases indicate that the scheme is better on the average than XMLPPM in terms of compression ratio, much faster in terms of running time and more economical in terms of memory usage. XMLPPM ran out of memory for UniProt data. The tool XMill runs much faster and with limited memory, but its average performance is considerably poor to that of XPrFAST as shown in Fig. 3 and Fig. 4.

The dynamic space requirements for the compressor **are** dominated by the size of the tables for the arithmetic compressor which grow exponentially with the size of the context. Also updating these tables after each symbol is processed makes the compression rather slow in comparison with dictionary based schemes.

6. ACKNOWLEDGMENTS

We wish to thank teachers of Department of Computer Science, Shivaji University, Kolhapur and Principal of Thakur College of Science and Commerce for motivating us for this research work.

7. REFERENCES

[1] Arion, A., Bonifati, A., Costa, G., D'Aguanno, S., Manolescu, I., Pugliese, A.: Efficient query evaluation over compressed XML data. In: EDBT. (2004) 200–218

[2] Backhouse, R.C.: Syntax of Programming Languages - Theory and Practice. Prentice Hall International, London (1979)

[3] Bzip2: (http://www.bzip.org)

[4] Cameron, R.D.: Source encoding using syntactic information source models. IEEE Transactions on Information Theory 34 (1988) 843–850

[5] Cleary, J.G., Teahan, W.J.: Unbounded length contexts for PPM. The Computer Journal 40 (1997) 67–75

[6] Cheney, J.: Compressing XML with Multiplexed Hierarchical PPM Models. In: Proceedings of the Data Compression Conference, IEEE Computer Society (2001) 163–172

[7] DBLP: (http://www.informatik.uni-trier.de/~ley/db)

[8] Ernst, J., Evans, W.S., Fraser, C.W., Lucco, S., Proebsting, T.A.: Code compression. In: PLDI. (1997) 358–365

[9] Franz, M.: Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile object systems. In: Mobile Object Systems: Towards the Programmable Internet. Springer-Verlag: Heidelberg, Germany (1997) 263–276

[10] Franz, M., Kistler, T.: Slim binaries. Commun. ACM 40 (1997) 87–94

[11] Fraser, C.W.: Automatic inference of models for statistical code compression. In: PLDI. (1999) 242–246

[12] Liefke, H., Suciu, D.: XMILL: An efficient compressor for XML data. In: SIGMOD Conference. (2000) 153–164

[13] Roza Leyderman, Oracle Database Sample Schemas, 11g Release 1 (11.1), B28328-03, Oracle, 2008.

[14] Min, J.K., Park, M.J., Chung, C.W.: XPRESS: A queriable compression for XML data. In: SIGMOD Conference. (2003) 122–133

[15] Nelson, M.: Arithmetic coding and statistical modeling.

[16] http://dogma.net/markn/articles/arith/part1.htm. Dr. Dobbs Journal (1991)

[17] Thierry Violleau, Java Technology and XML-Part One, March 2001, http://java.sun.com/developer/technicalArticles/xml/JavaTechandXML/

[18] Thierry Violleau, Java Technology and XML-Part Two, March 2002, http://java.sun.com/developer/technicalArticles/xml/JavaTechandXML_part2/

[19] Tolani, P.M., Haritsa, J.R.: XGRIND: A query-friendly XML compressor. In: ICDE. (2002) 225–234

[20] UniProt: (http://www.ebi.uniprot.org)

[21] Witten, I. H., Neal, R.M., Cleary, J.G.: Arithmetic coding for data compression. Commun. ACM 30 (1987) 520–540

[22] XML: W3C recommendation. http://www.w3.org/TR/REC-xml (2004)

[23] XMLZip, http://www.xmls.com

[24] Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23 (1977) 337–343

[25] Hariharan Subramanian and Priti Shankar, Compressing XML Documents Using Recursive Finite State Automata, CIAA 2005, LNCS 3845, pp. 282–293, 2006, Springer-Verlag Berlin Heidelberg 2006