

# Compact XML grammar based compression\*

S. Harrusi, A. Averbuch, A. Yehudai  
School of Computer Science  
Tel Aviv University, Tel Aviv 69978, Israel

## Abstract

Extensible Markup Language (XML) is the standard format for content representation and sharing on the Web. XML is a highly verbose language, especially regarding the duplication of meta-data in the form of elements and attributes. As XML content is becoming more widespread so is the demand to compress XML data volume.

This paper presents a new grammar, called D-grammar, which defines XML structure for a specific DTD. DTD is chosen as an explanatory example. The grammar can be extended to define other deterministic XML scheme languages such as XML Schema. It also presents a parser generator which generates a D-grammar parser. DPDT is an efficient and compact XML validator for the DTD which the D-grammar reflects. The presented compression technique encodes the DPDT validation choices during the XML structure parsing instead of the textual tags that compose the XML structure. This enhances the XML text compression twofold: first, there are less symbols to encode and second, the encoded structure symbols can predict the preceding text better than the textual structure tags.

A unique advantage of the presented technique is that it combines the validation phase with the compression phase and thus saves processing time.

This XML validation/compression fits streaming technologies and can be used in a wide variety of XML network applications such as gateways, routers, etc.

The DPDT validation choices are encoded by a partial prediction matching (PPM) codec, which is considered to be the state-of-the-art for text encoding.

We compare the performance of the presented algorithm, also called DPDT-L, with other existing XML compression techniques. The proposed compression algorithm achieves, on average, better compression ratio. The superiority of our compression technique is more evident when it is tested on XML medium size (~10MB) dataset.

## 1 Introduction

### 1.1 Motivation

Extensible Markup Language (XML) is the standard format for content representation (presentation) and sharing on the Web. Communication of information on machine level will ultimately be carried out

---

\*A preliminary version of the paper appeared in [50]

through XML. XML is a highly verbose language, especially regarding the duplication of meta-data in the form of elements and attributes. As the level of XML traffic grows so is the demand to compress XML data volume in order to reduce XML traffic bandwidth. XML on cellular communications networks [24] is a good example for the need to compress XML data.

Storing massive XML content before it is shared or presented on the Web is another need to have lossless XML compression. Again, the XML verbose nature significantly enlarges the volumes of the stored data.

It is clear that a lossless compression scheme for reducing XML volume is needed. In this paper, we treat XML in its most basic form - as a language. Each language has a grammar. Every grammar has a parser which recognizes it. But for XML languages, this assumption is not straightforward since there is no clear definition what is a XML parser. In the XML literature, the term XML parser actually means a lexical analyzer not a parser. There is no standard way to generate XML parsers for general purposes. There is also a difficulty to determine how to transform a syntactic XML dictionary into a formal grammar definition. We use the term syntactic dictionary to address the existing XML meta data description formats that contains DTD [34], XML-Schema [35], DSD [36], RELAX Core [37], TREX [38] and RELAX NG [39]. Our algorithm suggests how to generate automatically a XML parser according to a given dictionary. This XML parser generator can be used in a wide variety of XML applications such as validators, converters, editors, etc.

## 1.2 The basic idea

A lossless compression scheme for XML data is needed. This paper suggests a fully syntax based XML compression. We treat XML in its most general form - as a language whose underlying grammar is a variant of a context-free grammar (**CFG**). This is why we can benefit from twenty years of experience on the study of CFG source compression models and to implement and utilize a similar approach towards XML.

In the paper, we exploit the common form of syntactic dictionary to produce a new XML parsing technique. Our parser construction starts from a new grammar model, which we call a dictionary grammar (D-grammar). It is similar to a CFG with the following modifications:

1. Each non-terminal symbol appears as the right-hand-side of one production;
2. The left-hand-side of a production includes a regular expression that is enclosed by a unique pair of tagging characters.

This is a general approach towards XML manipulation. It creates a generic framework for XML processing. The XML parser accepts the D-grammar of documents described by this dictionary, which is the input dictionary. We call this process, which constitutes the core of this paper, XML parser generation. This framework is used to achieve XML compression.

The work in [2] suggested to use specific syntactic compressors that are planted inside the XML compression. When an XML document type is specified by a CFG, its definition can easily be expanded to include other CFG grammars. For example, if we want to syntactically encode URL addresses inside an XML document, we can expand the XML grammar with the URL grammar. URL address definition is even more restrictive than XML. It can be defined as a regular expression. The following regular expression (RE) illustrates the URL address structure:

$$\text{URL} ::= \text{'http://www.'} (\text{free-text '.'})? \text{free-text '.'} ( \text{'com'} | \text{'org'} ).$$

The 'free-text' is a predefined lexical symbol of the free text. Most of the structures that reside inside XML documents such as numbers, dates, IP addresses etc., will be compressed by XML lossless compression.

The proposed XML parser can be used for applications other than compression. The fact that this is a simple and fast generator of parsers, makes this parser generation technique very practical. Unlike common parsers that use prediction table for parsing, our XML parser uses a state machine instead of a table to determine the next production rule to be used for derivation. The state machine, which has a reduced number of states, serves as a compact prediction table. The parser takes into consideration the XML structure and its operation becomes efficient. The suggested XML parser generator can fit a wide variety of XML applications such as validators, converters, editors, etc (see [28]).

### 1.3 Outline of the lossless XML compression algorithm

The flow of the algorithm is given in Fig. 1. It contains three sub-modules:

1. **Dictionary conversion** - converts the dictionary to D-grammar (see section 3.3);
2. **XML parser generator** - creates an XML parser from its D-grammar;
3. **XML encoding** - encodes the XML parsers' moves.

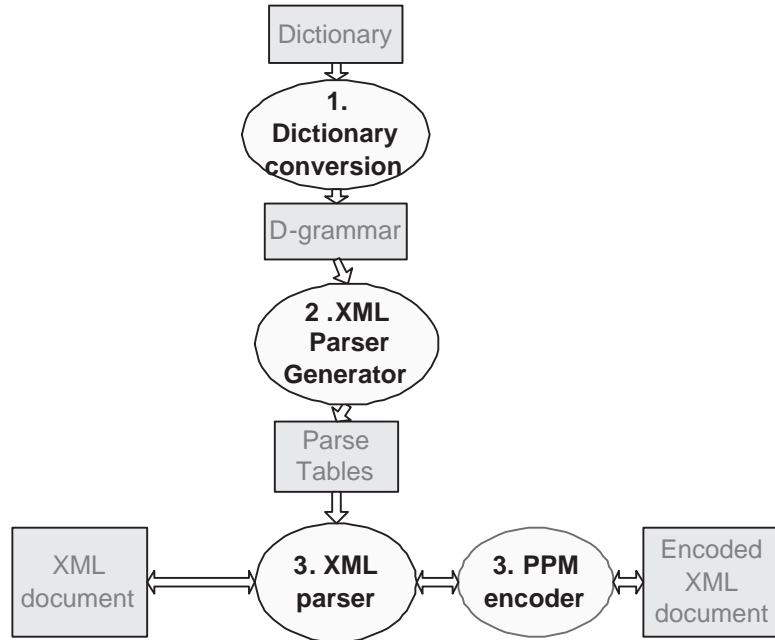


Figure 1: Flow of the XML lossless compression algorithm: the main components

Each element in the dictionary can be rephrased as a regular expression. This translation to D-grammar representation precedes the parser generation. We construct a Dictionary Deterministic Pushdown Transducer (DPDT) that acts as a parser for the given D-grammar (see section 3.3).

The third phase of the encoding algorithm uses the Partial Prediction Matching (**PPM** [11]), which is considered to be the state-of-the-art for text encoding. The encoder uses the XML parsing process to decide which are the lexical symbols that are relevant to the current elements' state. Only these symbols participate in the encoding process.

The decoder decodes the lexical symbols and sends them to the XML parser. The parser transforms it to its original XML format and writes it to a file.

A preliminary version of the basic DPDT-L algorithm was described in [50]. It neither provides the theoretical infrastructure nor updated benchmarks which are described here. This paper details the encoding scheme, formalizes the theory of DPDT generation and operation (see section 3) and new compression algorithms are applied to new benchmarks.

## 1.4 Main results

The comparison between the performance of our algorithm (DPDT-L) and the XMLPPM algorithm [9] is given in [50]. In this paper, we update and enhance the set of compression tools for which DPDT-L is compared against. Special emphasis is given to comparison with the compression techniques such as XMLPPM [9] and SCMPPM [44] that use the same PPM encoder as we do. We also compare with two DTD conscious encoders that are also based on PPM encoder: DTDPPM [45] and XAUST [46].

On average, our codec outperforms the other methods.

In [50] we evaluate the compression performance on small dataset (1MB). In this paper, the datasets are extended to medium (10MB) and large (100MB) sizes datasets.

The structure of the paper is as follows. Related compression and parsing algorithms are given in section 2. Section 3 describes the XML compression algorithm. The results after the completion of the application of the algorithm on standard benchmark datasets are given in section 4.

## 2 Related Works

In this section, we mention the main current XML compression methods. They are compared with our XML encoder design philosophy.

### 2.1 Context-Free-Grammar (CFG) encoding models

Over the past twenty years there have been attempts to find the best CFG encoding scheme. Two compression techniques emerged: the **derivational** and the **guided-parsing** techniques. The core of the derivational technique [14, 20, 18] is a step-by-step transmission of the derivation of a string from the goal symbol. At each step, the leftmost non-terminal is rewritten according to the grammar. Each non-terminal can only be rewritten by certain production rules. The derivational technique encodes the production rules choices.

The guided-parsing encoding method [13, 19, 16] is based on recording the moves a parser makes while parsing the text. Stone (and Al-Hussaini) choose LR(1) parsers for their broad coverage and thorough exploitation of grammatical information. Evans [19] applied it to both LR(1) and LL(1) parsers. Evans pointed out that the derivational metaphor is actually the same as the guided parsing metaphor, since e.g., the derivational method replays the LL(1) parser’s moves. In the rest of the paper, we refer to both techniques as **LL guided parsing** and **LR guided parsing** encoding methods.

Section 2.1.1 describes the LL guided-parsing encoding technique. We focus on this technique because it is the basis for our encoding method. Section 2.1.2 compares between LR guided parsing and LL guided parsing techniques. Section 2.1.3 describes how the guided-parsing encoding methods are used.

#### 2.1.1 LL guided parsing encoding models

The encoder in LL guided parsing, sends a series of production rules that derives the encoded string. The production rules series can be extracted from the LL(1) parsing process. Each time the top of the stack contains a non-terminal, a decision using a *decision table* is made on the next production rule to execute the derivation. LL guided-parsing encodes these decisions. We demonstrate the LL guided parsing encoding process on the XHTML document in Fig. 2. We use a single XHTML document (continues example) through this paper to demonstrate our encoding concepts. Figure 2 shows a

simple XHTML example document. Figure 2a shows the textual XML syntax of the example. Figure 2b illustrates how the XML document is represented on the WEB.

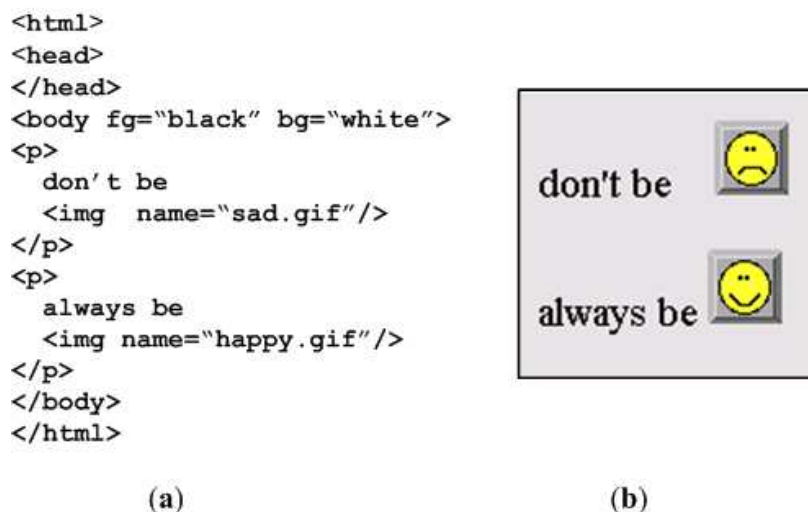


Figure 2: Example of an XHTML document. a) The XML syntax of the an XHTML document. It contains html tag ('<html >') with two nested tags: an empty header tag ('<head >') and a body tag ('<body >'). The body contains two paragraphs ('<p >'). Each paragraph contains text followed by an image tag ('<img >'). b) illustrates the WEB representation of this XHTML document.

Figure 3 shows the DTD of the XHTML example introduced in Fig. 2. This DTD defines a subset of the XHTML. We use this DTD to demonstrate our encoding principles. DTD is one example for an XML syntactic dictionary. It can be shown to fit XML Schema.

```

<!ELEMENT html head body >

<!ELEMENT head title? >

<!ELEMENT title #PCDATA >

<!ELEMENT body p* >
<!ATTLIST body
  fg (black | white) #REQUIRED
  bg (black | white) #IMPLIED
>

<!ELEMENT p (img | #PCDATA)* >

<!ELEMENT img >
<!ATTLIST img
  src #CDATA
  name #CDATA
>

```

Figure 3: DTD of the XHTML example introduced in Fig. 2. The DTD defines an XHTML subset. A `html` element (`'html'`) contains an header and a body elements. The header element (`'head'`) contains an optional `'title'` element. The (`'body'`) element contains multiple paragraph elements (`'p'`). Each paragraph element contains a mixture of image elements (`'img'`) and text elements (`'#PCDATA'`).

Figure 4 defines the *CFG* of an XHTML subset. We leave out the attributes definitions to simplify the presentation.

Index	Rule Production
PR.1	html => <html> head body </html>
PR.2	head => <head> title </head>
PR.3	head => <head> </head>
PR.4	title => <title> #PCDATA </title>
PR.5	body => <body> body_c </body>
PR.6	body_c => p body_c
PR.7	body_c =>
PR.8	p => <p> p_c </p>
PR.9	p_c => img p_c
PR.10	p_c => #PCDATA p_c
PR.11	p_c =>
PR.12	img => <img> </img>

Figure 4: A *CFG* definition of the XHTML subset that was declared in Fig. 3. Only the elements are defined in this grammar. A html element (**PR.1**) with an header and a body elements are defined. The header element (**PR.2-3**) has an optional title element (**PR.4**). The body element (**PR.5-7**) contains multiple paragraph elements (**PR.8-11**). Each paragraph contains a mixture of image elements (**PR.12**) and a free text.

The decision table in Fig. 4 grammar is given in Fig. 5.

	html	head	title	body	body_c	p	p_c	img
<html>	PR.1.							
<head>		PR. 2, 3						
<title>			PR. 4					
<body>				PR. 5				
</body>					PR. 7			
<p>					PR. 6	PR. 8		
<#PCDATA>							PR. 9	
<img>							PR. 10	PR. 12
</p>							PR. 11	

Figure 5: A decision table of the *CFG* that is defined in Fig. 4. Each terminal symbol that is a lookahead symbol defines a row. Each non-terminal symbol defines a column. When the LL-parser has a non-terminal symbol at the top of its stack, it extracts the production rule from the cell denoted by this non-terminal and the lookahead symbol.

The LL parsing process is illustrated in Fig. 6

Lookahead	Rule	Stack
<html>	PR. 1	html
		</html>, body, head, <html>
<head/>	PR. 3	</html>, body, head
		</html>, body, </head>, <head>
<body>	PR. 5	</html>, body
		</html>, </body>, body_c, <body>
<p>	PR. 6	</html>, </body>, body_c
		</html>, </body>, body_c, p
		</html>, </body>, body_c, </p>, p_c, <p>
"don't be"	PR. 10	</html>, </body>, body_c, </p>, p_c
		</html>, </body>, body_c, </p>, p_c, #PCDATA
<img>	PR. 9	</html>, </body>, body_c, </p>, p_c
	PR. 12	</html>, </body>, body_c, </p>, p_c, img
		</html>, </body>, body_c, </p>, p_c, </img>, <img>
</img>		</html>, </body>, body_c, </p>, p_c, </img>
</p>	PR. 11	</html>, </body>, body_c, </p>, p_c
		</html>, </body>, body_c, </p>
</body>	PR. 7	</html>, </body>, body_c
		</html>, </body>
</html>		</html>

Figure 6: The parsing process of the XHTML document that was defined in Fig. 2. The parser recognizes the grammar that is defined in Fig. 4. The **lookahead** column describes the lookahead terminal symbols. The **stack** column shows the contents of the stack during the parsing. Each cell shows the stack as a set of strings delimited by commas. The gray strings are terminal symbols and the black strings are non-terminals symbols. This stack symbol is the leftmost string in the top of the stack. When the top of the stack is a non-terminal symbol (black), the parser decides by using Fig. 5 decision table which production rule to apply. The **rule** column describes this production rule. This illustration is not complete. The second paragraph of the body element is missing. Its parsing is the same as the first paragraph. It applies the production-rules: PR.6, PR.10, PR.9, PR.12, PR.11 and PR.7.

The LL guided-parsing compression encodes the production rules choices which the LL parser applies. In the parsing example of Fig. 6, the rules column content is being encoded. The naive approach is to enumerate all the production rules globally and to use the global production number (**GPN**) [17] as the encoder's symbols. In the above example, the *GPN* of each production rule is its index, as appears in the index column in Fig. 4. The encoded symbols are:

**GPN:** PR.1, PR.3, PR.5, PR.6, PR.10, PR.9, PR.12, PR.11, PR.7.

The compression performance of *GPN* is not sufficiently good. Cameron [14] suggested to use a local production rule number (**LPN**) [17]. The LPN sequencing disposes wider determinism level. Each

non-terminal has a limited production set that can derive it. The production rules, in which it appears in the left side, are enumerated. The matched LPN number is encoded each time this non-terminal is derived. For example, when the decision table columns in Fig. 5 is examined, we see that there are three non-terminals which have a choice of multiple production rules: ‘*head*’, ‘*body\_c*’ and ‘*p\_c*’. We sort the production-rules of each non-terminal by its indices and enumerate them. For example, for the non-terminal ‘*head*’, the local enumeration is: 1(PR.2) and 2(PR.3). This enumeration is the local production number. The local encoded symbols of the above example are:

**LPN:** -, 2[2], -, 1[2], -, 2[3], 1[3], -, 3[3], -.

The ‘-’ character denotes a missing symbol that is encoded globally but not locally. The square brackets indicate the number of local enumerations that each symbol has.

### 2.1.2 LR vs. LL guided-parsing encoding models

LR guided parsing encoding is based on the information the parser has when a grammatical conflict occurs. Two types of conflicts are handled:

1. **Shift/Shift** - The encoder has to supply the lookahead symbol.
2. **Reduce/Reduce** - The encoder indicates the production rule.

The shift/reduction conflicts are not allowed in a legal LR grammar.

LR guided parsing exploits determinism whenever it occurs. The disadvantage of LR guided parsing is that during encoding top-down information is lost because of the bottom up nature of the LR parsing process. Because of its top down nature, the LL guided-parsing encoding exposes dependencies in the text that would otherwise remain hidden. Encoding of production rules implies that several terminals, which are part of the production rule derivation strings, are encoded by one symbol. But LL guided-parsing can also separate terminals by encoding the non-terminals in between neighboring terminals symbols. This phenomena is known as **order-inflation** [23]. Worse than order inflation, it even unclear whether additional non-terminals are needed. This phenomena is called **redundant categorization** [23]. Both phenomena, order inflation and redundant categorization, poorly affect the encoding quality. Our encoding algorithm is a top down in its nature. But it encodes terminals instead of production rules. The encoding of terminals prevent the order inflation and redundant categorization phenomena to occur.

### 2.1.3 Encoding methods for CFG models

A chronological view of related works identifies the evolution of encoding methods. In the 1980s, [16, 13, 15, 18] used Huffman coding to compress Pascal source-file corpus. In the late 1980, [14] targeted Pascal programs and used arithmetic coding. During the 1990s, programming languages have has been changed from Pascal to Java. [19] applied arithmetic coder to both Java and Pascal

sources. [21] applied LZW on Java files. [17] used PPM algorithm to reduce the size of Pascal sources. In recent years, the CFG compression goal has changed from compression of static archives to reduced throughput of dynamic XML and Java byte codes transmissions. [20] compressed Java mobile code with arithmetic coder. [22] adopted PPM for the same purpose. [9] encoded XML lexical symbols using a PPM algorithm. [23] used PPM to encode Scheme source code. Our encoding algorithm follows the trail of *CFG* source encoding methods and use *PPM* to encode the text in XML documents.

## 2.2 XML parsing

Current XML parsing theory is based upon regular tree grammars. In regular tree grammars, perspective XML documents are handled as textual representations of trees. Therefore, a dictionary specifies the structure of the trees. Various automata were introduced to implement tree grammars for XML parsing.

Three restrictive classes of regular tree grammars and their automata are defined in [40]. Each class defines and exposes the expressive power of a different XML schema language:

1. *Local tree* class defines the expressive power of the DTD schema language [34];
2. *Single type* class defines the expressive power of the W3C XML Schema language [35];
3. Regular tree grammar defines the expressive power of RelaxNG schema language [39].

Parsing of regular tree grammar is not deterministic. It may provides more than one interpretation of a document. As a result, its parsing time is not bounded by the length of the document. Therefore, it is impractical.

D-grammar parsing, which is described in section 3.3, is deterministic. Its expressive power equals to local tree class expressiveness. However, it can easily be adopted to express single class type of languages. Therefore, we can use most XML syntactic dictionaries that rely on this subclass: DTD [34], XML-schema [35] and deterministic RELAX NG [39] documents. Therefore, although the paper uses DTD as its underlying explanatory demonstration, it can fit other syntactic dictionaries.

### 2.2.1 Parsing of XML streams

Most of the proposed tree grammars automata have a major disadvantage. They are incapable to process XML streams. Neumann [33] constructed a top-down automata for regular tree grammars, which parses XML streams.

Our D-grammar automaton, called DPDT, fits to process XML streams. DPDT resembles the Neumann's automaton in its use of the regular expressions in the automata construction. But non-determinism complicates Neumann's automaton. Neumann's automaton has three sets of states. DPDT has a standard single set of states. This makes the DPDT construction more compact. Terminals participation in the production rules writings makes D-grammar a natural way to describe XML

attributes in particular and XML in general. In summary, we show that DPDT is a natural parser for XML documents.

### 2.2.2 XML validation

DTD validation of streaming XML documents under memory constraints was investigated in [48]. They showed the existence of an automaton with a bounded stack that is related to the depth of the XML document. This automaton has  $2^{|\Sigma|}$  states. DPDT also produces a strong XML validation. DPDT's automaton stack size is bounded by the depth of the XML document. The state space of the DPDT is more compact than the automaton in [48]. In most cases, the state space of the DPDT is linear in the size of the D-grammar.

The bounded stack size of the DPDT enhances the compression. It bounds the PPM context that predicts the encoded symbol. It makes D-grammar optimal for compression of XML documents that are grammar based.

## 2.3 XML Compression

XML compression is important mainly for two WEB applications: storage and transmission. The verbose nature of XML is disturbing for both. The static nature of storage usually allows it to use general encoders to achieve high compression ratios [2, 9, 26, 27, 41, 42, 43, 44, 45, 46]. XML database compressions have two variants: generic compression [2, 9, 27, 43, 44, 45, 46] and query enabling compression [26, 41, 42]. Query enabling compression takes into consideration a query mechanism which is applied to the stored XML data.

The encoding models in XML compression differ in several parameters:

1. The compressor can be either streaming or not.
2. They have different ways to compress the document's content and its structure. XML's content contains the text (*#CDATA* and *#PCDATA*) of the XML document. XML structure contains all the tags, attributes and special characters in the XML document. Cheney [47] defined two models for content encoding:
  - Multiplexed Hierarchical Modeling (MHM). The MHM approach switches among several PPM models.
  - Structural Context Modeling (SCM). In SCM, rather than switching among a small number of models that are based on the syntactic class of the data, the compressor uses a separate model to compress the content under each element symbol.
3. The encoding model in the XML compressors differs in the underlining encoding algorithm. It can utilize byte codes, LZW, Huffman, arithmetic coder and PPM.
4. How the compression exploits the structural information in the DTD.

This paper presents a new streaming compressor called DPDT-L. It is a generic database XML encoder that uses MHM approach with an underlying PPM encoder. The presented compressor switches between two PPM models: a structural model that encodes the XML validation decisions and a content model. Section 2.3.1 describes how other XML encoders model the structure and the content.

The DPDT-L encapsulates the structural information of the DTD in the validator operation. Several other encoding methods are also aware of the DTD structural information. Section 2.3.2 describes how different compressors exploit the structural information in the DTD.

### 2.3.1 XML encoding models

Transmission applications use byte codes to transfer the encoded source. It can be either a fixed byte-code [24, 25, 30] or a variable length byte-code [28, 29]. The most advanced encoding for transmission application was presented in [25, 29].

In order to be able to query the structure, most query enabled encoders separate structural compression from content compression. XMLzip [27] splits its content according to a certain depth of the XML tree structure and uses LZW to compress each sub-tree. XQueC [42] even separates between each path encoding. XQueC uses Huffman coding for encoding the structure and ALM for encoding the content. Xgrind [26] uses Huffman coding to encode the structure and arithmetic coding for encoding the content.

Generic database XML encoders use variety of encoding methods. XMill [2] splits the text of the XML document into containers and compresses each container using a text compressor such as gzip, bzip2 or PPM. XMill also uses semantic compressors to encode data items with a particular structure. The semantic compressors are based on a parser for a regular grammar.

XMLPPM [9] is a streaming compressor that uses an MHM encoding approach. The XMLPPM switches among several PPM models, one for element, attribute, character, and miscellaneous data, and “injects” element context symbols into the other models to recover lost accuracy due to model splitting. XMLPPM uses PPM as its underlying compressor.

SCMPPM [44] is a variant of the XMLPPM that uses the SCM encoding approach. AXECHOP [43] uses XMill’s container approach to encode text content and grammar based compression to encode the element structure of the document. XAUST [46] compressor takes advantage of the DTD information to compress the element structure and uses SCM encoding approach to compress the content (albeit using order-4 arithmetic coding rather than PPM). DTDPPM [45] is a DTD conscious extension of XMLPPM.

### 2.3.2 DTD awareness

The initial XML compression algorithms [2, 9, 27, 25] ignored the DTD information. Xcompress [31] and Xgrind [26] extract the list of expected elements from the DTD and encodes the index of the

element instead of the element itself. More sophisticated approach is used in the Millau project [29]. It creates a tree structure for each element that is specified in the DTD. The tree includes the relation to other elements, such as the special operator nodes for the regular expression operators that define the element content. The XML data is also represented as a tree structure. The DTD and the XML trees, are scanned in parallel and only the delta between the two representations is encoded. This method is called differential DTD. The same compression method was addressed more formally in [31]. Differential DTD does not extract the whole information from the DTD. Attribute definition of the DTD is not used by this method.

DTDPPM use of DTD is primarily to provide information about the element and attribute structure while supplying little information about text content. It removes whitespace from the XML document. The presented algorithm also removes whitespace from the XML document.

AXECHOP[43] generates a CFG that is capable of deriving this XML structure. This grammar is passed through an adaptive arithmetic coder before being written as a compressed file. The DPDT-L approach also generates a grammar that is capable of deriving this XML structure. But we use the D-grammar that is dedicated for describing the XML structure. A CFG description is too general for XML description.

XAUST[46] creates a FSM for each element in the DTD. The FSM describes the element content. In each encoding step, XAUST encodes the current element and the current state in the FSM of the element. The DPDT-L algorithm generalizes the XAUST algorithm. It combines the set of FSMs to a single automaton called DPDT. It encodes a single DPDT state in each encoding step instead the pairs  $\langle element, state \rangle$ . Furthermore, it encodes the state locally and not globally as XAUST does. This generalization enables the DPDT-L algorithm to combine the validation with the encoding process. The DPDT-L algorithm was developed independently of XAUST. The patent ([49]), which is based on the DPDT-L algorithm, was filed before the publication of XAUST.

## 2.4 Prediction by Partial Matching (PPM) encoding

A *context* is a finite length suffix of the current symbol. A *context model* is a conditional probability distribution over the alphabet that is computed from the contexts. *PPM* [11] is a finite context model encoding. The context model encoding uses the context model to predict the current symbol. The prediction is encoded and sent to the decoder. The context model is then updated by the current symbol and the encoding continues. A *finite context model* limits the length of contexts by which it predicts the current symbol. When the current context does not predict the current symbol, a special ‘**escape**’ event signals this fact to the decoder and the compression process continues with the context that is one event shorter. If zero length context does not predict the current symbol, the PPM uses an unconditional ‘order-1’ model as its baseline model.

We use in our encoding algorithm a variant of the *PPMD+* [12] that improves the basic PPM compression twofold: escape probability assignment and scaling. The ‘*D*’ escape probability assign-

ment method treats the escaping events as a symbol. When a symbol occurs it increments both the current symbol and the ‘escape’ symbol counts by  $1/2$ . ‘*D*’ method is generally used as the current standard method, due to its superior performance.

The ‘+’ term insinuates a **scaling** technique that the algorithm uses. Scaling here means distortion of the probabilities measurements in order to emphasis certain characteristics in the context. Two characteristics are scaled: if the current symbol was recently predicted in this context (**recent scaling**) or if no other symbol is predicted in this context (**deterministic scaling**).

The *PPMD+* algorithm uses arithmetic coder to encode its predicted symbols.

### 3 XML compression: the DPDT-L algorithm

The XML compression algorithm has two sequential components:

1. Generation of XML parser from its dictionary. Throughout the rest of the paper we use the DTD as an illustrative example of a dictionary. The same works for XML Schema and others.
2. XML compressor that uses the parser from the first component.

In the first component, the dictionary is converted into a set of regular expressions (RE). Each XML element is described as a single RE - see section 3.1. Then, an XML parser is generated from this description in the following way. A Deterministic Pushdown Transducer, which produces a leftmost parse, is generated - see section 3.3. This parser is similar to a LL parser. The output of the parser - namely the leftmost parse - is used as an input to the guided parsing compressor, which constitutes the second component of the algorithm - see section 3.6.

The guided parsing compression has three components:

1. The XML tokenizer accepts the XML source and outputs lexical tokens;
2. The XML parser parses the lexical tokens;
3. The PPM encodes the lexical symbols using information from the parser.

The algorithm’s flow is given in Fig. 7. The vertical flow describes the sequential stages. The horizontal flow describes the iterative parsing and the encoding process. Two parsers, XML parser (**3b** in Fig. 7) and the parser’s generation (**2c** in Fig. 7) operate independently. They contain the same iterative process.

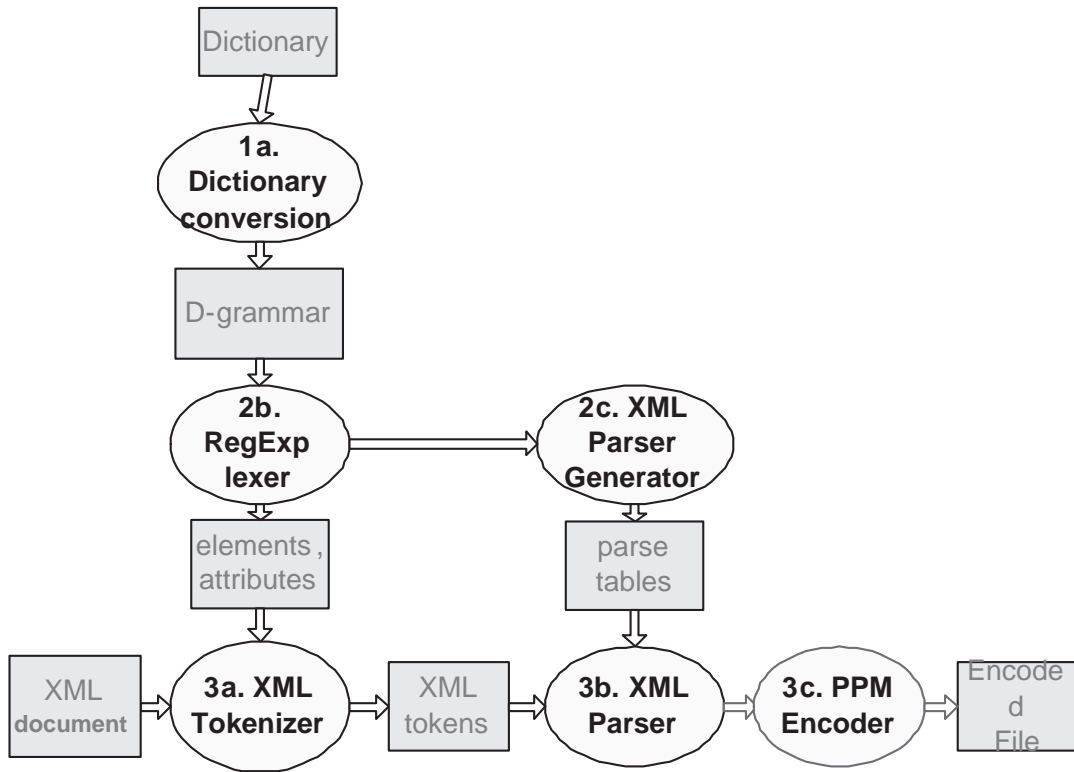


Figure 7: Flow of the XML compression algorithm (DPDT-L)

In the next sections, we give detailed descriptions for various components in the XML compression algorithm as they appear in Fig. 7.

### 3.1 Dictionary conversion

We describe now the flow of **1a** (dictionary conversion) in Fig. 7. The dictionary is translated into a set of REs. An XML element is described as a concatenation of a start tag string, attributes list, the element’s content and the end tag string. The RE syntax is given as:

“<element” attributes “>” element-content “</element>” .

Figure 8 describes the RE description of the XHTML subset. The RE is converted from the original DTD (Fig. 8a). The attributes are described as a concatenation of the pair attribute and value. Implied attributes are described with the optional operator character ‘?’. Text free attribute values are described with the reserved string CDATA. A selection of attribute values is described as in the DTD. Figure 8b shows all the attributes that were converted to RE:

1. The ‘src’ attribute of the ‘img’ element is an explicit attribute with a free text value. Its RE conversion is ‘src CDATA’.

2. The 'name' attribute of the 'img' element is an implicit attribute with a free text value. Its RE conversion is '? (name CDATA)'.
3. The 'text' attribute of the 'body' element is an explicit attribute with selection of the values 'black' or 'white'. Its RE conversion is 'text (black|white)'.

The reserved PCDATA string is used for free text elements. See for example the title element content.

<pre> &lt;!ENTITY color (black white)&gt; &lt;!ENTITY code #CDATA&gt;  &lt;!ELEMENT html head body &gt;  &lt;!ELEMENT head title? &gt;  &lt;!ELEMENT title #PCDATA &gt;  &lt;!ELEMENT body p* &gt; &lt;!ATTLIST body   text %color   background %color &gt;  &lt;!ELEMENT p (img   #PCDATA)* &gt;  &lt;!ELEMENT img p*&gt; &lt;!ATTLIST img   src %code   name %code </pre>	<pre> "&lt;html&gt;"   head body "&lt;/html&gt;"  "&lt;head&gt;"   title? "&lt;/head&gt;"  "&lt;title&gt;"   PCDATA "&lt;/title&gt;"  "&lt;body" text (black white)   background (black white) "&gt;"   p* "&lt;/body&gt;"  "&lt;p&gt;"   (img   PCDATA)* "&lt;/p&gt;"  "&lt;img" src CDATA name CDATA "&gt;"   p* "&lt;/img&gt;" </pre>
---	--

Figure 8: DTD conversion of XHTML subset. Left: DTD description of its HTML subset. Right: Regular expression description of the HTML subset

### 3.2 The RE lexer

We describe now the flow of **2b** (RegExp lexer) in Fig. 7. The RE has three tokens types: 1. RE operator's characters; 2. XML reserved character; 3. Textual tokens.

The following RE operators exist: 1. Parenthesis: (,); 2. Multiplication: +,\*; 3. Optional: ?; 4. And: &; 5. Or: |.

The XML reserved character `>` marks the end of element character. It distinguishes between elements and attributes to enable the tokenizer to determine which symbol to produce.

The RE lexer has three functions: 1. Tokenizes a regular expression; 2. Generates a lexical symbol from tokens; 3. Classifies textual token by its XML entity types which are element, attribute and attribute's value.

A state machine with three states is being used to tokenize the RE (see Fig. 9). Each state fits a different XML entity type. Each token is replaced with a lexical symbol. The lexical symbol is given to the XML parser generator as an input symbol. It is saved in the lexer for a future use by the next analyzed tokens and by the XML lexer. The XML lexer inherits its symbols' table from the RE lexer. The XML entity type, which is known according to the current lexer state, is also saved. The XML entity type will be used by the XML lexer (see section 3.4) in order to correctly represent a decoded token.

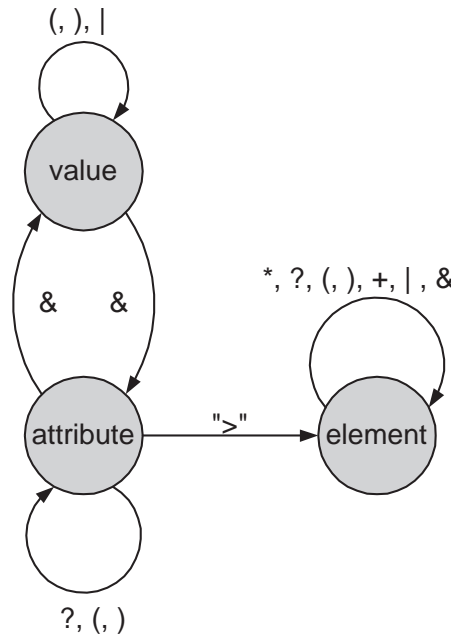


Figure 9: Finite state machine for RE lexer

### 3.3 Parser Generator

This section presents the parsing algorithm of an XML file. Note that we use the term parsing as it appears in Computer Science literature (e.g. Formal Language Theory, Compilers, etc.). This is in contrast to the use of the term parsing in some of the XML literature, as noted in Section 2.2.

We rely on the fact that the dictionaries of an XML file constitute an Extended Backus Normal Form (EBNF) grammar for the rest of the file. EBNF grammars are not strictly CFGs, because they use some form of regular expressions in the the right-hand-side of their productions. On the other hand, each XML element is delimited by a unique pair of start tag and end tag (in angled brackets).

This fact is used to simplify the parsing process.

For example, ‘<html>’ is the right bracket of the first RE in Fig. 8 and ‘</html >’ is the left bracket. None of them appear elsewhere in the grammar.

In our presentation, we will consider the special form for a dictionary grammar, which we call D-grammar. We assume that the reader is familiar with the basics of Automata, Language and Parsing Theory ([32]). Its notation is adopted here.

**Definition 3.1.** A D-grammar is a 4-tuple  $G = (N, \Sigma, P, A_1)$  where  $N = \{A_1, A_2, \dots, A_n\}$  is a finite non-empty set of non-terminals,  $\Sigma$  is a finite non-empty set of terminal symbols, divided between two disjoint subsets  $\Sigma = \{a_1, \bar{a}_1, a_2, \bar{a}_2, \dots, a_n, \bar{a}_n\} \cup \Sigma'$  where  $\Sigma'$  is a collection of attributes.  $A_1$  is the start symbol, and  $P$  is a non empty set of bracketed productions, with the following form: each non-terminal  $A_i$  has a unique production  $A_i \rightarrow a_i R_i \bar{a}_i$ , where  $a_i, \bar{a}_i \in \Sigma$  are the *left and right bracket* for  $A_i$ , respectively, and  $R_i$  is a regular expression over  $N \cup \Sigma'$  (we will call it  $A_i$ 's regular expression). Note that the brackets of different non-terminals are distinct.

For example, in the grammar of Figure 8,  $N = \{\mathbf{html}, \mathbf{head}, \mathbf{title}, \mathbf{body}, \mathbf{p}, \mathbf{img}\}$ ,  $A_6 = \mathbf{img}$ ,  $a_6 = \mathbf{<img}$ ,  $\bar{a}_6 = \mathbf{</img >}$ , and  $R_6 = \mathbf{src CDATA name CDATA >p *}$ .

A D-grammar is used to derive words in  $\Sigma^*$  by repeatedly applying production to a non-terminal symbol. This is similar to the way a CFG is used, except that the right hand side of a production is not a fixed word, like in a CFG, so when a production of  $A_i \rightarrow a_i R_i \bar{a}_i$  of a D-grammar is applied to  $A_i$ ,  $A_i$  is replaced by an arbitrary word  $a_i \beta \bar{a}_i$ , such that  $\beta \in R_i$ .

More formally, we define

**Definition 3.2.** Let  $G = (N, \Sigma, P, A_1)$  be a D-grammar. We define the relation  $\Rightarrow$  (read “derives”) on words over  $N \cup \Sigma$  as follows. If  $A \in N$ ,  $\alpha, \gamma \in (N \cup \Sigma)^*$ ,  $A \rightarrow a R \bar{a} \in P$  and  $\beta \in R$ , then  $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ . We will also say that  $\alpha A \gamma \Rightarrow \alpha \beta \gamma$  *uses the production*  $A \rightarrow a R \bar{a} \in P$ . If  $\alpha \in \Sigma^*$ , then we call the derivation *leftmost*, and denote it by  $\alpha A \gamma \Rightarrow_L \alpha \beta \gamma$ . (Henceforth we will be interested only in leftmost derivations). We use the usual notation for the reflexive transitive closure of the derives relation to indicate derivation of any length: If  $\delta_0 \Rightarrow_L \delta_1 \Rightarrow_L \dots \Rightarrow_L \delta_m$  for some  $m \geq 0$ , then we write  $\delta_0 \Rightarrow_L^* \delta_m$ .

Further, if for each  $j, 1 \leq j \leq m - 1$ ,  $\delta_j \Rightarrow_L \delta_{j+1}$  uses production  $A_{i_j} \rightarrow a_{i_j} R_{i_j} \bar{a}_{i_j} \in P$ , then the *leftmost parse* of the derivation  $\delta_0 \Rightarrow_L^* \delta_m$  is the sequence of production numbers  $i_0 i_1 \dots i_{m-1}$  which we will denote  $\pi(\delta_0 \Rightarrow_L^* \delta_m)$ .

The language defined by a non-terminal symbol  $A_i$ , is  $L(A_i) = \{w \in \Sigma^* | A_i \Rightarrow_L^* w\}$ . The language defined by the grammar is simply the language defined by the start symbol  $A_1$ .

We will now show how to construct a Deterministic Pushdown Transducer (DPDT) that acts as a parser for the given D-grammar. A DPDT is a pushdown automaton with output. First we present a definition of a DPDT adapted from [32], but simplified: For our purpose, we need not be concerned with  $\epsilon$  moves.

**Definition 3.3.** A ( $\epsilon$  free) Deterministic Pushdown Transducer, (henceforth simply DPDT) is a 8-tuple  $M = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$  where  $Q$  is a finite set of *states*,  $\Sigma$  is a finite *input alphabet*,  $\Gamma$  is a finite *pushdown alphabet*,  $\Delta$  is a finite *output alphabet*,  $\delta$  is a function from  $Q \times \Sigma \times \Gamma$  to  $Q \times \Gamma^* \times \Delta^*$  called the *transition function*,  $q_0 \in Q$  is the *initial state*,  $Z_0$  is the *initial stack symbol*, and  $F \subseteq Q$  is the set of *final* or *accepting states*.

A *configuration* of  $M$  is a 4-tuple  $(q, w, \gamma, v)$  in  $Q \times \Sigma^* \times \Gamma^* \times \Delta^*$ , where  $q$  is the current state of  $M$ ,  $w$  is the unread portion of the input,  $\gamma$  is the content of the stack, (its leftmost symbol is the top of the stack), and  $v$  is the output produced so far.

A move of  $M$  is represented by a relation  $\vdash$  between configurations, defined as follows:  $(q, aw, Z\alpha, v) \vdash (p, w, \gamma\alpha, vu)$  if  $\delta(q, a, Z) = (p, \gamma, u)$ , for some  $q, p \in Q, a \in \Sigma, w \in \Sigma^*, Z \in \Gamma, \gamma, \alpha \in \Gamma^*$  and  $v, u \in \Delta^*$ .

We use  $\vdash^*$  to denote a computation of any length.

A word  $w$  is accepted by  $M$  and *translated* into  $v$  if  $(q_0, w, Z_0, \epsilon) \vdash^* (q, \epsilon, \epsilon, v)$  for some  $p \in F$ : when  $M$  is started in its initial state, with the stack containing the initial symbol, and with  $w$  in its input, it terminates in a final state, with an empty stack, having consumed all its input, and produced  $v$  as its output.

We will now present the DPDT  $M$  that is constructed to act as a parser for a given D-grammar. Given a word  $w \in \Sigma^*$ , if  $w$  is generated by the D-grammar, then given  $w\$$  as input, (where  $\$$  is a special end marker),  $M$  will read the input to completion, terminate in an accepting state and empty the stack, and produce as output the leftmost parse  $\pi(A_1 \Rightarrow_L^* w)$ . Otherwise the DPDT will reject  $w\$$  - it will not terminate as described.

The construction of  $M$  is defined as follows.

**Definition 3.4.** Let  $G = (N, \Sigma, P, A_1)$  be a D-grammar, and let  $M_0, M_1, M_2, \dots, M_n$  be Finite State Automata (FSA), so that for  $i \geq 1$ ,  $M_i$  accepts the language  $R_i$ ,  $A_i$ 's regular expression. The FSA  $M_0$  is added to simplify the construction. It accepts the language  $\{A_1\}$ .

In particular,  $M_i = (Q_i, N \cup \Sigma', \delta_i, q_{0i}, F_i)$ . For  $M_0$ , specifically,  $Q_0 = \{q_{00}, f_0\}$ ,  $F_0 = \{f_0\}$ ,  $\delta_0(q_{00}, A_1) = f_0$  and  $\delta_0$  is undefined elsewhere. We assume, without loss of generality, that the sets of states  $Q_i$  are disjoint.

We now define a DPDT as follows:  $M = (Q, \Sigma \cup \{\$\}, \Gamma, \Delta, \delta, q_{00}, Z_0, \{f_0\})$  where  $Q = \bigcup_{i=0}^n Q_i$ ,  $\Gamma = \{Z_0\} \cup \{[q, a_i] | q \in Q, 0 \leq i \leq n\}$ . The output alphabet  $\Delta = \{1, 2, \dots, n\}$  represents production numbers. The transition function  $\delta$  has four types of rules, depending on the type of input symbol:

**Type 1** For all  $1 \leq i \leq n, 0 \leq j \leq n, Z \in \Gamma$  and  $q \in Q_j$ , we have  $\delta(q, a_i, Z) = (q_{0i}, [\delta_j(q, A_i), a_i]Z, i)$  (left bracket).

**Type 2** For all  $1 \leq i \leq n, q \in Q$ , and  $p \in F_i$ , we have  $\delta(p, \bar{a}_i, [q, a_i]) = (q, \epsilon, \epsilon)$  (right bracket).

**Type 3** For all  $0 \leq i \leq n, q \in Q_i, a \in \Sigma'$  and  $Z \in \Gamma$ , we have  $\delta(q, a, Z) = (\delta_i(q, a), Z, \epsilon)$  (non bracket symbol).

**Type 4**  $\delta(f_0, \$, Z_0) = (f_0, \epsilon, \epsilon)$  (end marker).

$\delta$  is undefined for all other values of its arguments.

In the sequel, we will use  $\vdash^i$  (and  $\vdash^{i*}$ ) to denote a computation step (sequence of steps) of type  $i$ .

It can easily be seen that  $M$  is deterministic, and has no  $\epsilon$  moves.

$M$  operates as follows. When given non bracket symbols,  $M$  simulates the behavior of an individual FSM in its state, each time following a word  $\beta$  to see if it belongs to a specific  $R_j$  (type 3 moves). Whenever a left bracket  $a_i$  appears in the input, the DPDT must suspend its simulation of the current FSM  $M_j$ , pushing onto the stack a symbol that combines the state  $q \in Q_j$  from which this simulation is to be resumed later (explained below), and the left bracket  $a_i$ .  $M$  then starts a simulation of the regular expression  $R_i$  by changing its state to the initial state  $q_{0i}$  of the corresponding FSM  $M_i$  (type 1 move). Whenever a right bracket  $\bar{a}_i$  is read,  $M$  must be in an accepting state  $p \in F_i$  of the current FSM being simulated  $M_i$ . Further, the right bracket being read  $\bar{a}_i$  must match the left bracket  $a_i$  on the stack. If these conditions hold, then the stack symbol  $[q, a_i]$  is popped and the simulation resumes from the state  $q \in Q_j$  (type 2 move).

The state  $q \in Q_j$  from which simulation is to be resumed (which is pushed onto the stack along with the right bracket) is computed as follows. The right bracket  $a_i$  that causes suspension uniquely determines the non-terminal symbol  $A_i$  for which a derivation step is considered. When the simulation of  $M_i$  is completed in an accepting state, and followed by the appearance of  $\bar{a}_i$  in the input, this corresponds to completion of the right hand side of the production  $A_i \rightarrow a_i R_i \bar{a}_i$ . As far as the FSM  $M_j$ , whose operation have been suspended, this amounts to viewing the symbol  $A_i$ , so the state in which the simulation should be resumed should be  $\delta_j(q, A_i)$ , where  $q$  was the state in which the simulation of  $M_j$  was suspended. (This justifies the definition of a type 1 move).

One can see that the DPDT traverses the derivation tree left to right, top down. It moves down when processing left brackets (type 1), right when processing non bracket symbols (type 3), and up when processing right brackets (type 2). It pushes a symbol on the stack while going down, and pops a symbol while going up. It produces an output symbol only when it goes down – it outputs the production number  $i$  when reading  $a_i$ . After reading a word  $w \in A_1$ ,  $M$  will be in its accepting state, and the stack will contain the initial stack symbol only. Reading the end marker will now empty the stack (type 4), terminating the computation successfully. One can see that if the computation terminates successfully, the resulting output is exactly the left parse of the input word.

We demonstrate the DPDT operation on the XHTML introduced in section 2. Figure 10 illustrates the FSA ( $M_i$ ) constructed from the DTD of Fig. 3.

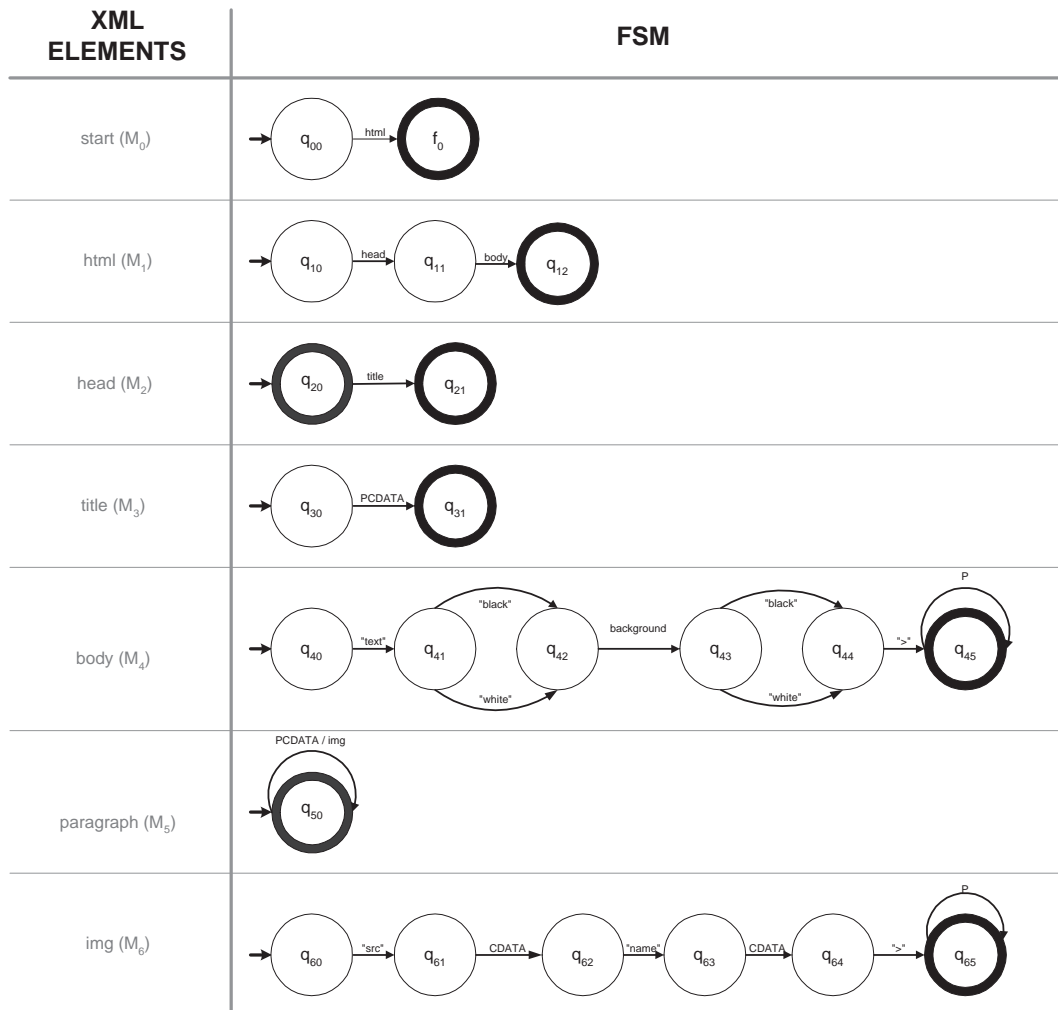


Figure 10: The FSA that accepts the XHTML elements in Fig. 8 is constructed from the RE. There are seven *FSA*, one for each of the six non-terminals ( $M_1$ - $M_6$ ), and  $M_0$  which is used to start the transcoding. The circles are states of the FSA. Accepting states are denoted by a thick circle, while start states are denoted by an incoming arrow.

Figure 11 describes the DPDT operation.

Lookahead	Type	State (Q)	Output	Stack
<html>	1	q <sub>00</sub>	q <sub>00</sub>	Z <sub>0</sub>
<head>	1	q <sub>10</sub>	q <sub>10</sub>	[f <sub>0</sub> , <html>] , Z <sub>0</sub>
</head>	2	q <sub>20</sub>	q <sub>20</sub>	[q <sub>11</sub> , <head>] , [f <sub>0</sub> , <html>] , Z <sub>0</sub>
<body	1	q <sub>11</sub>	q <sub>11</sub>	[f <sub>0</sub> , <html>] , Z <sub>0</sub>
text	3	q <sub>40</sub>	q <sub>40</sub>	[q <sub>12</sub> , <body>] , [f <sub>0</sub> , <html>] , Z <sub>0</sub>
....				
>	3	q <sub>44</sub>	q <sub>44</sub>	[q <sub>12</sub> , <body>] , [f <sub>0</sub> , <html>] , Z <sub>0</sub>
<p>	1	q <sub>50</sub>	q <sub>50</sub>	[q <sub>45</sub> , <p>] , [q <sub>12</sub> , <body>] , [f <sub>0</sub> , <html>] , Z <sub>0</sub>
"don't be"	3	q <sub>50</sub>	q <sub>50</sub>	[q <sub>45</sub> , <p>] , [q <sub>12</sub> , <body>] , [f <sub>0</sub> , <html>] , Z <sub>0</sub>
<img	1	q <sub>60</sub>	q <sub>60</sub>	[q <sub>50</sub> , <img>] , [q <sub>45</sub> , <p>] , [q <sub>12</sub> , <body>] , [f <sub>0</sub> , <html>] , Z <sub>0</sub>
....				
>	3	q <sub>64</sub>	q <sub>64</sub>	[q <sub>50</sub> , <img>] , [q <sub>45</sub> , <p>] , [q <sub>12</sub> , <body>] , [f <sub>0</sub> , <html>] , Z <sub>0</sub>
</img>	2	q <sub>65</sub>	q <sub>65</sub>	[q <sub>50</sub> , <img>] , [q <sub>45</sub> , <p>] , [q <sub>12</sub> , <body>] , [f <sub>0</sub> , <html>] , Z <sub>0</sub>
</p>	2	q <sub>50</sub>	q <sub>50</sub>	[q <sub>45</sub> , <p>] , [q <sub>12</sub> , <body>] , [f <sub>0</sub> , <html>] , Z <sub>0</sub>
</body>	2	q <sub>45</sub>	q <sub>45</sub>	[q <sub>12</sub> , <body>] , [f <sub>0</sub> , <html>] , Z <sub>0</sub>
</html>	2	q <sub>12</sub>	q <sub>12</sub>	[f <sub>0</sub> , <html>] , Z <sub>0</sub>
\$	4	f <sub>0</sub>	f <sub>0</sub>	Z <sub>0</sub>
		f <sub>0</sub>		

Figure 11: DPDT parsing of the XHTML document which appears in Figure 2. The table contains five columns. The *lookahead* lexical symbol, the transition *type* (1-4), the current transcoder *state* and the current *stack* content and the output.

The proof that the DPDT indeed works as expected, will proceed by proving a series of lemmas:

The first lemma shows how to partition a derivation tree into its top production and a collection of subtrees.

**Lemma 3.1.** *Let  $w$  be a word in  $a_i \Sigma^* \bar{a}_i$  for some  $i, 1 \leq i \leq n$ . Then  $w \in L(A_i)$  if and only if  $w$  can be partitioned as  $w = a_i x_1 y_1 x_2 y_2 \dots x_k y_k x_{k+1} \bar{a}_i$  for some  $k \geq 0$ , such that*

- for all  $1 \leq j \leq k + 1, x_j \in \Sigma'^*$
- For all  $1 \leq j \leq k, y_j \in L(A_{i_j})$  for some  $A_{i_j} \in N$ , and
- $\hat{w} = x_1 A_{i_1} x_2 A_{i_2} \dots x_k A_{i_k} x_{k+1} \in R_i$ ,

Furthermore,  $\hat{w}$  is uniquely determined from  $w$ .

*Proof.* If  $w \in L(A_i)$ , then there must be a derivation  $A_i \Rightarrow_L a_i \hat{w} \bar{a}_i \Rightarrow_L^* w$ , such that  $\hat{w} \in R_i$ . Furthermore, since  $\hat{w}$ , has no bracket symbols (by the definition of a the regular expressions in a

D-grammar), there is a unique way to decompose around its  $k \geq 0$  non-terminal symbols,  $\hat{w} = x_1 A_{i_1} x_2 A_{i_2} \dots x_k A_{i_k} x_{k+1}$ , where  $x_j \in \Sigma'^*$  for  $1 \leq j \leq k+1$ , and  $A_{i_j} \in N$  for  $1 \leq j \leq k$ . So the derivation  $a_i \hat{w} \bar{a}_i \Rightarrow_L^* w$  can be rewritten as

$$a_i x_1 A_{i_1} x_2 A_{i_2} \dots x_k A_{i_k} x_{k+1} \bar{a}_i \Rightarrow_L^* a_i x_1 y_1 x_2 y_2 \dots x_k y_k x_{k+1} \bar{a}_i$$

where for each  $j, 1 \leq j \leq k+1$ ,  $A_{i_j} \Rightarrow_L^* y_j$ .

The other direction is trivial. □

Next, we show how the DPDT simulates a single FSA on a string of non brackets that belongs to some  $L(A_i)$ .

**Lemma 3.2.** *For all  $i, 1 \leq i \leq n, x \in \Sigma'^*, Z \in \Gamma$ ,*

1. *If there exists  $z$ , such that  $xz \in R_i$ , then  $(q_{0i}, x, Z, \epsilon) \vdash^* (\delta_i(q_{0i}, x), \epsilon, Z, \epsilon)$*
2. *If  $(q_{0i}, x, Z, \epsilon) \vdash^* (p, \epsilon, \gamma, v)$  for some  $p \in Q, \gamma \in \Gamma^*$ , and  $v \in \Delta^*$  then  $p = \delta_i(q_{0i}, x), \gamma = Z, v = \epsilon$  and the derivation uses type 3 moves only.*

*Proof.* Each direction may be proved by a straightforward induction on the length of  $x$ , omitted. □

We can now show that each word derived from a non-terminal induces a certain computation of  $M$ .

**Lemma 3.3.** *For all  $1 \leq i \leq n, q \in Q, Z \in \Gamma$  and  $w \in L(A_i)$*

$$(q, w, Z, \epsilon) \vdash^* (\delta_l(q, A_i), \epsilon, Z, \pi(A_i \Rightarrow_L^* w))$$

where  $q \in Q_l$ .

*Proof.* We will prove the lemma by induction on the height of the derivation tree.

**Basis:** The height of the derivation tree is 1. Then  $w \in L(A_i)$  implies that  $w = a_i x_1 \bar{a}_i$ ,  $x_1 \in \Sigma'^*$ ,  $\hat{w} = x_1 \in R_i$  and  $A_i \rightarrow a_i R_i \bar{a}_i \in P$ . By construction of  $M$ , for all  $l, 1 \leq l \leq n, q \in Q_l$

$$(q, a_i x_1 \bar{a}_i, Z, \epsilon) \stackrel{1}{\vdash} (q_{0i}, x_1 \bar{a}_i, [\delta_l(q, A_i), a_i] Z, i) \stackrel{3}{\vdash}^* (\delta_i(q_{0i}, x_1), \bar{a}_i, [\delta_l(q, A_i), a_i] Z, i) \stackrel{2}{\vdash} (\delta_l(q, A_i), \epsilon, Z, i)$$

We used Lemma 3.2 for the middle part of the computation (type 3 moves). The last step (type 2 move) is valid since  $x_1 \in R_i$  implies that  $\delta_l(q_{0i}, x_1) \in F_i$ .

To complete the basis, we just note that  $i = \pi(A_i \Rightarrow_L^* a_i R_i \bar{a}_i)$ .

**Induction step:** Assume the lemma holds for all  $w'$  and all  $i'$  such that the height of the derivation tree for  $A_{i'} \Rightarrow_L^* w'$  is at most  $h$  for some  $h > 0$ . Now assume  $A_i \Rightarrow_L^* w$  with a derivation tree of height  $h + 1$ . By Lemma 3.1 the derivation can be rewritten as

$$A_i \Rightarrow_L a_i x_1 A_{i_1} x_2 A_{i_2} \dots x_k A_{i_k} x_{k+1} \bar{a}_i \Rightarrow_L^* a_i x_1 y_1 x_2 y_2 \dots x_k y_k x_{k+1} \bar{a}_i$$

where for each  $j, 1 \leq j \leq k + 1$ ,  $A_{i_j} \Rightarrow_L^* y_j$ . Furthermore, the derivation trees of all  $A_{i_j} \Rightarrow_L^* y_j$ , have height at most  $h$ , so we can use the induction hypothesis for each of them.

In order to complete the proof of the induction step, we need the following claim.

**Lemma 3.4.** *Let  $w = a_i x_1 y_1 x_2 y_2 \dots x_m y_m x_{m+1}$ , such that  $x_j \in \Sigma'^*$  for  $1 \leq j \leq m + 1$ ,  $A_{i_j} \Rightarrow_L^* y_j$ , for all  $1 \leq j \leq m$ , and assume that Lemma 3.3 holds for these derivations. Let  $\hat{w} = x_1 A_{i_1} x_2 A_{i_2} \dots x_m A_{i_m} x_{m+1}$ , and suppose there exists  $z$  such that  $\hat{w}z \in R_i$ . Then for all  $Z \in \Gamma$*

$$(q, w, Z, \epsilon) \vdash^* (\delta_i(q_{0i}, \hat{w}), \epsilon, [\delta_l(q, A_i), a_i]Z, i\pi(A_{i_1} \Rightarrow_L^* y_1)\pi(A_{i_2} \Rightarrow_L^* y_2) \dots \pi(A_{i_m} \Rightarrow_L^* y_m))$$

*Proof.* The proof will be by induction on  $m$ .

**Basis:**  $m = 0$ . Then  $w = a_1 x_1, \hat{w} = x_1 \in \Sigma'^*$  and there exists  $z$  such that  $x_1 z \in R_i$ . Then by construction, for any  $q \in Q, Z \in \Gamma$ ,  $(q, a_1 x_1, Z, \epsilon) \stackrel{1}{\vdash} (q_{0i}, x_1, [\delta_l(q, A_i), a_i]Z, i)$  where  $q \in Q_l$ . Further, by Lemma 3.2 we get

$$(q_{0i}, x_1, [\delta_l(q, A_i), a_i]Z, i) \stackrel{3}{\vdash} (\delta_i(q_{0i}, x_1), \epsilon, [\delta_l(q, A_i), a_i]Z, i)$$

which completes the basis.

**Induction step:** Suppose the claim holds for all  $m < m_0$ , for some  $m_0 > 0$ . Now let  $m = m_0$ . Let  $w = a_i x_1 y_1 x_2 y_2 \dots x_m y_m x_{m+1}$ , such that  $x_j \in \Sigma'^*$  for all  $1 \leq j \leq m + 1$ ,  $A_{i_j} \Rightarrow_L^* y_j$ , for all  $1 \leq j \leq m$ , and assume that Lemma 3.3 holds for these derivations. Suppose there exists  $z$ , such that  $\hat{w}z \in R_i$  where  $\hat{w} = x_1 A_{i_1} x_2 A_{i_2} \dots x_m A_{i_m} x_{m+1}$ . Let  $w_1 = a_i x_1 y_1 x_2 y_2 \dots x_{m-1} y_{m-1} x_m$ . By the induction hypothesis for all  $Z \in \Gamma$

$$(q, w_1, Z, \epsilon) \vdash^* (\delta_i(q_{0i}, \hat{w}_1), \epsilon, [\delta_l(q, A_i), a_i]Z, i\pi(A_{i_1} \Rightarrow_L^* y_1)\pi(A_{i_2} \Rightarrow_L^* y_2) \dots \pi(A_{i_{m-1}} \Rightarrow_L^* y_{m-1}))$$

Since  $w = w_1 y_m x_{m+1}$ , we can write

$$(q, w_1 y_m x_{m+1}, Z, \epsilon) \vdash^* (\delta_i(q_{0i}, \hat{w}_1) y_m x_{m+1}, [\delta_l(q, A_i), a_i]Z, i\pi(A_{i_1} \Rightarrow_L^* y_1) \dots \pi(A_{i_{m-1}} \Rightarrow_L^* y_{m-1}))$$

We now consider derivation  $A_{i_m} \Rightarrow_L^* y_m$ , and use Lemma 3.3 to extend  $M$ 's computation as follows:

$$(\delta_i(q_{0i}, \hat{w}_1)y_m x_{m+1}, [\delta_l(q, A_i), a_i]Z, i\pi(A_{i_1} \Rightarrow_L^* y_1) \dots \pi(A_{i_{m-1}} \Rightarrow_L^* y_{m-1})) \vdash^*$$

$$(\delta_i(\delta_i(q_{0i}, \hat{w}_1), A_i), x_{m+1}, [\delta_l(q, A_i), a_i]Z, i\pi(A_{i_1} \Rightarrow_L^* y_1) \dots \pi(A_{i_{m-1}} \Rightarrow_L^* y_{m-1})\pi(A_{i_m} \Rightarrow_L^* y_m))$$

We now use Lemma 3.2 and apply the equation  $\delta_i(\delta_i(q, u_1), u_2) = \delta_i(q, u_1 u_2)$  twice to extend the computation further

$$(\delta_i(q_{0i}, \hat{w}_1 A_i), x_{m+1}, [\delta_l(q, A_i), a_i]Z, i\pi(A_{i_1} \Rightarrow_L^* y_1)\pi(A_{i_2} \Rightarrow_L^* y_2) \dots \pi(A_{i_m} \Rightarrow_L^* y_m)) \vdash^*$$

$$(\delta_i(q_{0i}, \hat{w}_1 A_i x_{m+1}), \epsilon, [\delta_l(q, A_i), a_i]Z, i\pi(A_{i_1} \Rightarrow_L^* y_1)\pi(A_{i_2} \Rightarrow_L^* y_2) \dots \pi(A_{i_m} \Rightarrow_L^* y_m))$$

This establishes the entire computation, and completes the proof of the induction step. Thus, lemma 3.4 has been established. □

We can now complete the induction step in the proof of Lemma 3.3. Consider again the word  $w = a_i x_1 A_{i_1} x_2 A_{i_2} \dots x_k A_{i_k} x_{k+1} \bar{a}_i$  and the derivation

$$A_i \Rightarrow_L a_i x_1 A_{i_1} x_2 A_{i_2} \dots x_k A_{i_k} x_{k+1} \bar{a}_i \Rightarrow_L^* a_i x_1 y_1 x_2 y_2 \dots x_k y_k x_{k+1} \bar{a}_i$$

where for each  $j, 1 \leq j \leq k+1$ ,  $A_{i_j} \Rightarrow_L^* y_j$ . Let  $w = w' \bar{a}_i$ . Then the conditions of Lemma 3.4 apply to  $w' = a_i x_1 y_1 x_2 y_2 \dots x_k y_k x_{k+1}$ , (with  $z = \epsilon$ ) and from the lemma we get the computation

$$(q, w, Z, \epsilon) \vdash^* (\delta_i(q_{0i}, \hat{w}), \bar{a}_i, [\delta_l(q, A_i), a_i]Z, i\pi(A_{i_1} \Rightarrow_L^* y_1)\pi(A_{i_2} \Rightarrow_L^* y_2) \dots \pi(A_{i_m} \Rightarrow_L^* y_m))$$

By definition, the leftmost parse of a derivation is the production used in its first step, followed by the leftmost parses of the subtrees from left to right. Hence

$$i\pi(A_{i_1} \Rightarrow_L^* y_1)\pi(A_{i_2} \Rightarrow_L^* y_2) \dots \pi(A_{i_m} \Rightarrow_L^* y_m) = \pi(A_i \Rightarrow_L^* w)$$

Also, since  $\hat{w} \in R_i$ ,  $\delta_i(q_{0i}, \hat{w}) \in F_i$ , the computation may be extended by

$$(\delta_i(q_{0i}, \hat{w}), \bar{a}_i, [\delta_l(q, A_i), a_i]Z, \pi(A_i \Rightarrow_L^* w)) \stackrel{2}{\vdash} (\delta_l(q, A_i), \epsilon, Z, \pi(A_i \Rightarrow_L^* w))$$

This completes the induction step and the entire proof of Lemma 3.3. □

The next Lemma is the converse of Lemma 3.3

**Lemma 3.5.** *If  $(q, w, Z, \epsilon) \vdash^* (p, \epsilon, Z, v)$  for some  $q, p \in Q, Z \in \Gamma$ , and  $v \in \Delta^*$  so that all intermediate configurations in this computation have stack height larger than 1, then there exist  $i$  and  $l$ , such that*

$1 \leq i \leq n, 0 \leq l \leq n, w \in L(A_i), q \in Q_l, p = \delta_l(q, A_i)$ , and  $v = \pi(A_i \Rightarrow_L^* w)$ .

*Proof.* Since all intermediate configurations in this computation have stack height larger than 1, it follows that the first step must be a type 1 move, and the last step a type 2 move. So  $w = a_i x_1 \bar{a}_i$ . Let  $q \in Q_l$ , for some  $0 \leq l \leq n$ , and let  $p = \delta_l(q, A_i)$ .

We proceed by an induction on the maximal stack height during the computation. **Basis:** The maximal stack height is 2, so the computation can be written as

$$(q, a_i x_1 \bar{a}_i, Z, \epsilon) \stackrel{1}{\vdash} (q_{0i}, x_1 \bar{a}_i, [p, a_i]Z, i) \stackrel{3}{\vdash^*} (p'_1, \bar{a}_i, [p, a_i]Z, i) \stackrel{2}{\vdash} (p, \epsilon, Z, i)$$

where  $p'_1 = \delta_i(q_{0i}, x_1)$  (by Lemma 3.2), and  $p'_1 \in F_i$  (to allow for the type 2 move). Clearly also  $i = i'$ . It follows that  $x_1 \in R_i$ , so that  $w = a_i x_1 \bar{a}_i \in L(A_i)$  with  $\pi(A_i \Rightarrow_L^* w) = i$  (a single step derivation). This completes the basis.

**Induction step:** Assume the lemma holds for computations of maximal stack height less than  $h$ , for some  $h > 2$ . Now consider a computation with maximal stack height  $h$ .

Since the height of the stack can be changed by at most 1 in each step, we can identify the longest subcomputations that occur at a fixed stack height of 2, and decompose the computation as follows, using the fact that moves that do not change the stack height are of type 3, which do not change the content of the stack and do not produce output. As in the basis, the left and right bracket symbols must match, so one can write  $w = a_i x_1 y_1 x_2 y_2 \dots x_k y_k x_{k+1} \bar{a}_i$  and decompose the computation as

$$\begin{aligned} & (q, a_i x_1 y_1 x_2 y_2 \dots x_k y_k x_{k+1} \bar{a}_i, Z, \epsilon) \stackrel{1}{\vdash} (p_1, x_1 y_1 x_2 y_2 \dots x_k y_k x_{k+1} \bar{a}_i, [p, a_i]Z, i) \stackrel{3}{\vdash^*} \\ & (p'_1, y_1 x_2 y_2 \dots x_k y_k x_{k+1} \bar{a}_i, [p, a_i]Z, i) \stackrel{3}{\vdash^*} (p_2, x_2 y_2 \dots x_k y_k x_{k+1} \bar{a}_i, [p, a_i]Z, i v_1) \stackrel{3}{\vdash^*} \\ & (p'_2, y_2 \dots x_k y_k x_{k+1} \bar{a}_i, [p, a_i]Z, i v_1) \stackrel{3}{\vdash^*} \dots \stackrel{3}{\vdash^*} (p_{k+1}, x_{k+1} \bar{a}_i, [p, a_i]Z, i v_1 v_2 \dots v_k) \stackrel{3}{\vdash^*} \\ & (p'_{k+1}, \bar{a}_i, [p, a_i]Z, i v_1 v_2 \dots v_k) \stackrel{2}{\vdash} (p, \epsilon, Z, i v_1 v_2 \dots v_k) \end{aligned}$$

where intermediate configuration in the subcomputations on the words  $y_j$  have stack height larger than 2, so they are not dependent on the actual stack symbols. Hence we can say that for all  $1 \leq j \leq k$  and  $Z' \in \Gamma$   $(p'_j, y_j, Z', \epsilon) \vdash^* (p_{j+1}, \epsilon, Z', v_j)$ , where the maximal stack height of these computations is less than  $h$ . The type 1 move (the first step in the derivation) implies that  $p_1 = q_{0i}$ .

Applying the induction hypothesis to the computations  $(p'_j, y_j, Z', \epsilon) \vdash^* (p_{j+1}, \epsilon, Z', v_j)$  for all  $1 \leq j \leq k$ , we get that  $y_j \in L(A_{i_j}), p'_j \in Q_{l_j}, p_{j+1} = \delta_{l_j}(p'_j, A_{i_j}), v_j = \pi(A_{i_j} \Rightarrow_L^* y_j)$ . Looking at the type 3 subcomputations, we get from Lemma 3.2, that  $p'_j = \delta_i(p_j, x_j)$  for all  $1 \leq j \leq k$ . In addition, since each of the type 3 subcomputations is followed by a type 1 move (the computations on  $y_j$  start by increasing the size of the stack), we must have  $p'_j \in F_{i_j}$ .

By combining all the above, we can see that all  $l_j$  are identical, and equal to  $l$ .

for all  $1 \leq j \leq k$ .  $v_j = \pi(A_{i_j} \Rightarrow_L^* y_j)$ . Hence  $i v_1 v_2 \dots v_k = i \pi(A_{i_1} \Rightarrow_L^* y_{i_1}) \dots \pi(A_{i_k} \Rightarrow_L^* y_{i_k}) =$

$\pi(A_i \Rightarrow_L^* w)$ .

□

**Theorem 3.6.** *Given a D- grammar, one can construct a DPDT  $M$  that works as follows. For each  $w \in \Sigma^*$ ,  $M$  accepts  $w$  if and only if  $w \in L(A_1)$ . Furthermore, if  $w \in L(A_1)$ , then  $M$  produces as output the left parse of  $w$ .  $M$  has no  $\epsilon$  moves, so its running time is linear in the length of  $w$ .*

*Proof.* Follows from Lemma 3.3 and Lemma 3.5.

If  $w \in L(A_1)$  then by Lemma 3.3  $(q_{00}, w, Z_0, \epsilon) \vdash^* (f_0, \epsilon, Z_0, \pi(A_i \Rightarrow_L^* w))$ , since  $\delta_0(q_{00}, A_1) = f_0$ . Adding the end marker, and a type 4 move we get  $(q_{00}, w\$, Z_0, \epsilon) \vdash^* (f_0, \$, Z_0, \pi(A_i \Rightarrow_L^* w)) \vdash (f_0, \epsilon, \epsilon, \pi(A_i \Rightarrow_L^* w))$ .

Conversely, if  $w\$$  is accepted by  $M$ , then its computation must be of the form

$$(q_{00}, w\$, Z_0, \epsilon) \vdash^* (f_0, \$, Z_0, v) \stackrel{4}{\vdash} (f_0, \epsilon, \epsilon, v).$$

We can now use Lemma 3.5, noting that  $q_{00} \in Q_0$ ,  $f_0 = \delta_0(q_{00}, A_1)$  and  $\delta_0$  is undefined elsewhere, to conclude that  $w \in L(A_1)$ , and  $v = \pi(A_1 \Rightarrow_L^* w)$ .

The linear running time follows from the construction of  $M$  as  $\epsilon$  free.

□

We can therefore construct a parser generator, that constructs the parsing tables (a variation of the DPDT shown above) while reading the dictionary portion of the XML file. Then, the parser is applied to the rest of the XML file, producing the leftmost parse as explained (see Section 3.5).

The size of the parser (the number of states) may, in the worst case, be exponential in the size of the original grammar, because the construction involves conversion of non-deterministic FSA to deterministic FSA. However, in practice, we can expect, the parser is not much larger than the original grammar. The running time of the parser's generator may therefore be exponential in the worst case, but it is linear in practice. In any event, the running time of the parsing is linear in the size of the input.

### 3.4 XML lexer

The flow **3a** (XML tokenizer) in Fig. 7 is described now. The XML lexical analyzer (lexer) inherits its symbols table from the RE lexer. The table maps symbols to XML tokens. The XML lexer reads XML tokens from a XML source. It retrieves its matched lexical symbol from the symbol table and sends it to the XML parser. The lexer uses two types of predefined symbols: Free text element is wrapped with the PCDATA lexical symbol, and free text attribute value is wrapped with the CDATA lexical symbol. Figure 12 illustrates the XML lexer state machine. It has five states to determine which string is currently tokenized: start tag or end tag or attribute or free text attribute value or selection list attribute value.

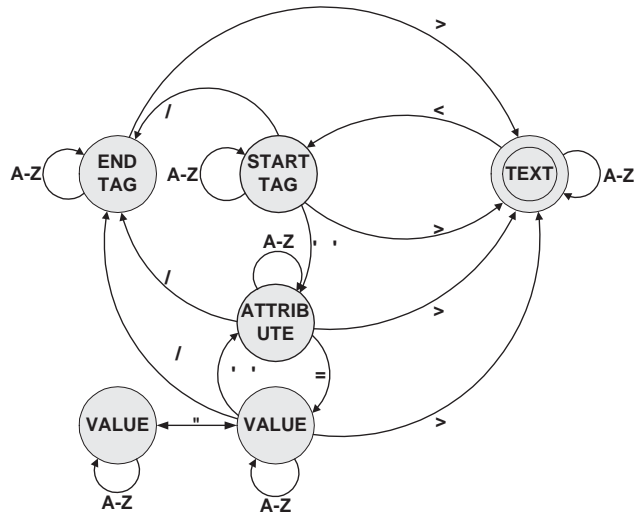


Figure 12: The XML lexer state machine

The XML lexer also supplies a reverse functionality. It receives a lexical symbol from the decoder and writes the matched XML token to the output XML source. In order to represent the token correctly it must know its XML entity type. The XML entity type of each symbol is inherited from the RE lexer as part of the symbol table. The following XML representation occurs in the decoding process:

**attribute:** attribute =

**start element:** <element>

**end element:** </element>

**attribute value:** "value"

### 3.5 The DPDT parser

We describe now the flow of **3b** (XML parser) in Fig. 7. The DPDT, generated as described in section 3.3, is applied to the stream of XML tokens, producing the leftmost parse as explained. Since the DPDT has no  $\epsilon$  moves, it works in linear time. (Its operation is similar to the LL parser operation - working top down with no backtracking).

As noted in section 3.3, the output of the DPDT is the left parse of the input word, namely, a list of the production numbers used in the parse tree, listed top down, left to right.

### 3.6 DPDT guided encoding

The DPDT-L encoding method multiplexes the content model encoding and the structure model encoding using the same PMM model. The structure model symbols are the DPDT finite output

alphabet symbols  $\Delta$ . The DPDT-L algorithm executes the DPDT on the input XML document and encode the output symbols  $a \in \Delta$ . Its encoding is locally guided by the DPDT. Section 2.1.1 describes local LL-guided-parser encoding that encodes the relevant production rules. Relevant production rules can derive the non-terminal at the top of the stack.

The DPDT guided encoding, encodes the output symbols instead of production rules. Local DPDT guided encoding, encodes the DPDT output symbols that are relevant for the current DPDT state. The relevant DPDT output symbols are determined by the DPDT transition function. Each transition type assigns a relevancy type symbol as follows:

**Type 1:** For all  $1 \leq i \leq n, 0 \leq j \leq n$  and  $q \in Q_j$ , if  $\delta_j(q, A_i)$  is defined, then  $a_i$  is **relevant** to  $q$  (left bracket).

**Type 2:** For all  $1 \leq i \leq n$  and  $q \in F_i$ ,  $\bar{a}_i$  is **relevant** to  $q$  (right bracket).

**Type 3:** For all  $0 \leq i \leq n, q \in Q_i, a \in \Sigma'$ , if  $\delta_i(q, a)$  is defined, then  $a$  is **relevant** to  $q$  (non-bracket symbol).

A single relevant symbol is ignored by the encoding algorithm. In the XHTML example, the relevant symbols are shown in Fig. 13. It is constructed from the REs in Fig. 10.

State (Q)	Relevant Symbol <sub>[type]</sub>
$q_{20}$	<code>&lt;/head&gt;</code> <sub>[2]</sub> , <code>&lt;title&gt;</code> <sub>[3]</sub>
$q_{41}, q_{43}$	<code>black</code> <sub>[3]</sub> , <code>white</code> <sub>[3]</sub>
$q_{45}$	<code>&lt;p&gt;</code> <sub>[1]</sub> , <code>&lt;/body&gt;</code> <sub>[2]</sub>
$q_{50}$	<code>&lt;img</code> <sub>[1]</sub> , <code>&lt;/p&gt;</code> <sub>[2]</sub> , <code>pcdata</code> <sub>[3]</sub>
$q_{65}$	<code>&lt;p&gt;</code> <sub>[1]</sub> , <code>&lt;/img&gt;</code> <sub>[2]</sub>

Figure 13: Table of XHTML relevant symbols which are constructed from the transitions in Fig. 10. The list of relevant symbols is detailed for each state. The square brackets to the right of each symbol mark its relevancy-type.

When the XHTML example in Fig. 11 is encoded, we receive the following local encoded symbols:

- , - , </head > , - , - , ... , - , <p > , “don’t be” , <img , ... , - , - , </p > , </body > , -

The character ‘-’ marks deterministic lexical symbols that are ignored by the encoder. The ‘...’ marks the places in the example where the parsing details are not shown.

Implementation of a local DPDT encoding by PPMD+ is straightforward. PPMD+ implementation uses an exclusion bit mask that refers to symbols that are excluded during the symbol encoding

process (see section 2.4). Normally, the PPMD+ initializes an empty exclusion mask for every encoded symbol. In local DPDT encoding, when a symbol is encoded, we mark the non-relevant symbols in the exclusion mask. Thus, the PPMD+ encoder ignores the non-relevant symbols and only the relevant symbols are encoded.

The content encoding model is plain. It has 255 ASCII symbols. All character symbols are relevant for encoding of an ASCII symbol.

## 4 XML compression results

In this section, we evaluate the performance of the encoding algorithm that was introduced in section 3. In section 4.1, the XML benchmarks documents (XML corpus) used in the experiment are described. The performance of the compression algorithm (DPDT-L) is compared in section 4.2 to other XML compression methods.

### 4.1 The source data

Our XML corpus contains four available XML benchmarks files [51, 52, 53, 54]. The different benchmarks are XML documents with a range of distinct structural characteristics. The XML corpus has different sizes: large ( $\sim 100\text{MB}$ ), medium ( $\sim 10\text{MB}$ ) and small ( $\sim 1\text{MB}$ ).

Table 1 provides structural information of the XML corpus.

Document	Structure (%)	Average depth	Average freedom
Xmark	28	5	3
007	78	9	1
Michigan	17	14	2
XMach-1	14	5	4

Table 1: Characteristics of the benchmark files that are used by the DPDT-L compression algorithm. Column 2 (Structure) is the number (in percentage) of characters in the dataset that are XML tags. The average depth of the stack in the parser is given in column 3 (Average depth). The average number of relevant symbols (Average freedom) is given in column 4.

Table 1 contains an additional statistics that was gathered by the DPDT-L algorithm during the parsing of the XML documents such as the average XML tree depth (Average depth). Relevant symbols were introduced in section 3.6. Relevant symbols are symbols that are accepted by the current DPDT state. The average number of relevant symbols (Average freedom) is given in column 4.

The XML corpus contains four documents. The characteristics of these documents (datasets) are: **XMark** benchmark document [53]. The XMark data generator produces XML documents modeling an auction website, which is a typical e-commerce application.

**007** benchmark document [51]. The x007 data generator produces structure centric synthetic XML documents with a simple structure.

**Michigan** benchmark document [52]. The Michigan data generator produces content centric XML documents with a highly redundant content and a simple structure.

**XMach-1** benchmark document [54]. The XMach-1 data generator produces synthetic XML documents. It randomly creates its content from a given set of words.

## 4.2 Compression ratios

The current XML compression methods to which we compare the performance of the DPDT-L were described in section 2.3. We compare the presented DPDT local encoding schemes (**DPDT-L**) to existing methods that use the same underlying PPM encoder. **DTDPPM** [45] and **XAUST** [46] are DTD aware encoders. **XMLPPM** [9] and **SCMPPM** [44] ignore the DTD.

Table 2 summarizes the bytes per character (bpc) computed by different compression methods on small sizes ( $\sim 1\text{MB}$ ) datasets.

File	Size (KB)	DPDT-L	XAUST	DTDPPM	XMLPPM	SCMPPM
Xmark	1155	1.63	1.75	1.87	1.61	1.68
007	1085	0.10	0.18	0.20	0.18	0.14
Michigan	909	0.61	0.63	0.60	0.61	0.54
XMach-1	1091	2.11	2.12	2.28	2.11	2.09
Average		1.143	1.204	1.275	1.160	1.145

Table 2: bpc from different compression methods operated on small datasets ( $\sim 1\text{MB}$ ). The average is weighted.

Table 3 summarizes the bpc from different compression methods applied to medium sizes ( $\sim 10\text{MB}$ ) datasets.

File	Size (KB)	DPDT-L	XAUST	DTDPPM	XMLPPM	SCMPPM
Xmark	11,597	1.40	1.48	1.85	1.39	1.54
007	13,510	0.23	0.37	0.27	0.45	0.21
Michigan	10,308	0.56	0.59	0.58	0.57	0.50
XMach-1	16,513	1.78	1.85	2.28	1.78	2.09
Avarage		1.052	1.132	1.323	1.106	1.162

Table 3: bpc from different compression methods operated on medium sizes ( $\sim 10\text{MB}$ ) datasets

Table 4 summarizes the bpc from different compression methods applied to large sizes ( $\sim 100$ MB) datasets.

File	Size(KB)	DPDT-L	XAUST	DTDPPM	XMLPPM	SCMPPM
Xmark	115,775	1.28	1.38	1.86	1.27	1.51
007	130,992	0.41	0.44	0.34	0.55	0.27
BM	102,907	0.55	0.58	0.61	0.55	0.50
Xmach-1	85,554	1.68	1.80	2.28	1.68	2.09
Avarage		0.924	0.988	1.188	0.967	1.009

Table 4: bpc from different compression methods applied to large sizes ( $\sim 100$ MB) datasets

Table 5 summarizes the improvements in % using DPDT-L algorithm in comparison with other compression methods. This is achieved by taking the weighted average, which is related to the file size, of the compression ratios.

Size (KB)	XAUST	DTDPPM	XMLPPM	SCMPPM
Small	5.02	10.36	1.46	0.16
Medium	7.13	20.50	4.94	2.56
Large	6.46	28.49	4.31	1.73

Table 5: Improvements in % of the DPDT-L against different compression methods.

The results in Table 5 clearly show that local DPDT encoded guided parser (DPDT-L algorithm) outperforms, on average, the rest. Both DTD aware compressors achieve lower CR on every file in the corpus. DPDT-L improves XAUST CR by 6% on average. 25% on the average is the improvement over the performance of the DTDPPM algorithm.

The results are less evident on DTD unaware compressors. The content of the XML files in the corpus can be roughly divided into: simple and complex. Simple content is highly redundant (Michigan) or its content portion in the file is redundant in comparison to the structure portion of the file (007). Files with simple content are best compressed by SCMPPM. However, files with complex content (XMach-1 and XMark) are best compressed by MHM based encoders such as XMLPPM and DPDT-L.

On the average, XMLPPM and SCMPPM achieve similar CR. The advantage and disadvantage of each compression method are balanced.

MHM based compression methods such as DPDT-L and XMLPPM, converge to similar CR on complex content. But DPDT-L CR is better (up to 50%) on files with simple content files. On the average, this advantage makes DPDT-L CR better than XMLPPM and SCMPPM methodologies.

The results show that medium size ( $\sim 10$ MB) files achieve better CR improvement when DPDT-L is used .

## 5 Future work

This research can be extended into the following directions:

- No special attention was paid to efficient and standard implementation of the DPDT-L XML compression algorithm. We plan to replace the propriety XML lexical analyzer with a fast and standard XML parser. The PPM model should be replaced by a dynamic allocation mechanism in order to reduce memory utilization.
- From understanding the advantages of SCM based encoders, it is useful to combine the DPDT-L encoding structure with the SCM content encoding.

## References

- [1] ISO 8879, Information Processing–Text and Office Systems–Standard Generalized Markup Language (SGML), 1986.
- [2] H. Liefke and D. Suci, “XMill: an efficient compressor for XML data”, Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 153- 164, 2000.
- [3] D. Connolly and J. Bosak, “Extensible Markup Language(XML)”, W3C Activity Group page, 1997.
- [4] D. Raggett, A. Le Hors and I. Jacobs, “HTML 4.0 Specification”, World Wide Web Consortium Working Draft, September 1997.
- [5] Barry, Paoly, McQueen, and Maler, “Extensible HyperText Markup Language, A Reformulation of HTML 4.0 XML 1.0”, <http://www.w3.org/TR/2000/REC-xhtml1-20000126>.
- [6] <http://www.informatik.uni-trier.de/~ley/db/index.html>
- [7] <http://www.ibiblio.org/xml/examples/>
- [8] <http://www.tpc.org/>
- [9] J. Cheney, “Compressing XML with multiplexed hierarchical models”, Data Compression Conference (DCC 2001), Snowbird, Utah, Proceedings of IEEE, pp. 163–172, 2001.
- [10] W. J. Teahan, PPMD+, PPM\* source code, <http://www.cs.waikato.ac.nz/~wjt/>.
- [11] J.G. Cleary, I.H. Witten, “Data compressing using adaptive coding and partial string matching”, IEE Trans. Comm., 32(4) pages 396-402, 1984.

- [12] Teahan. W.J. and Cleary J.G, “The entropy of english using PPM based models”, Data Compression Conference (DCC), Snowbird, Utah, Proceedings of IEEE, pp. 53–62,
- [13] A.M.M. Al-Hussaini, “File compression using probabilistic grammars and LR parsing”, PhD thesis, Loughborough University, 1983.
- [14] R. D. Cameron, “Source encoding using syntactic information source models”, IEEE Transactions on Information Theory, 34(4), 843-850, July 1988.
- [15] J. F. Contla, “Compact coding of syntactically correct source programs”, Software-Practice and Experience, 15(7), 625-636, 1985.
- [16] R. G. Stone, “On the choice of grammar and parser for the compact analytical encoding of programs”, Computer Journal, 29(4), 307-314, 1986.
- [17] J. Tarhio, “Context coding of parse trees”, Data Compression Conference (DCC 1995), Snowbird, Utah, Proceedings of IEEE, pp. 442, 1995.
- [18] J. Katajainen, M. Penttonen, and J. Teuhola, “Syntax-directed compression of program files”, Software-Practice and Experience, 16(3), 269-276, 1986.
- [19] W.S. Evans, “Compression via guided parsing”, Storer and Cohn [181], page 544. 1998.
- [20] P. Eck, X. Changsong and R. Matzner, “A new compression scheme for syntactically structured messages (programs) and its applications to Java and the Internet”, Data Compression Conference (DCC 1998), Snowbird, Utah, Proceedings of IEEE, pp. 542, 1998.
- [21] M. Franz, “Code-Generation On-the-Fly: A Key to Portable Software”, PhD thesis, ETH Zurich, 1994.
- [22] C.H. Stork, V. Haldar, and M. Franz, “Generic adaptive syntax-directed compression for mobile code”, Technical Report 00-42, Department of Information and Computer Science, University of California, Irvine, Nov. 2000.
- [23] M. Lake, “Prediction by grammatical match”, Data Compression Conference (DCC 2000), Snowbird, Utah, Proceedings of IEEE, pp. 153–162, 2000.
- [24] “WAP Binary XML Content Format”, W3C NOTE 24 June 1999.
- [25] M. Girardot, N. Sundaresan, “Millau: an encoding format for efficient representation and exchange of XML over the Web”, Proceedings of the 9 th WWW Conference, May 2000, Amsterdam, Netherlands.
- [26] P.M. Tolani, J.R. Haritsa, “XGRIND: A Query-friendly XML Compressor”, Database Systems Lab, SERC Indian Institute of Science Bangalore 560012, India, 2001

- [27] XMLSolutions Corporation, XMLZip. <http://www.xmlzip.com/>, 1999.
- [28] J. Jeuring, and P. Hagg, “Generic Programming for XML Tools”, Institute of Information and Computing Sciences, Utrecht University, The Netherlands, May 2002.
- [29] N. Sundaresan, R. Moussa, “Algorithms and programming models for efficient representation of XML for Internet applications”, International World Wide Web Conference 2001, 366-375, 2001
- [30] U. Niedermeier, J. Heuer, A. Hutter, W. Stechele, “MPEG-7 Binary Format for XML Data”, Data Compression Conference (DCC 2001), Snowbird, Utah, Proceedings of IEEE, pp. 53–62, 2001.
- [31] M. Levene and P. Wood, “XML Structure Compression”, Birkbeck College, University of London, London, U.K., 2002
- [32] A. V. Aho and J. D. Ullman, “The Theory of Parsing, Translation, and Compiling”, Vol. I, Prentice Hall, 1972.
- [33] A. Neumann, “Parsing and Querying XML Documents in SML”, PhD thesis, University of Trier, Trier, 2000.
- [34] T. Bray, J. Paoli, and C. M. Sperberg McQueen (Eds), “Extensible Markup Language (XML) 1.0 (2nd Edition)”, W3C Recommendation, Oct. 2000, <http://www.w3.org/TR/2000/REC-xml-20001006>
- [35] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn (Eds), “XML Schema Part 1: Structures”, W3C Recommendation, May 2001, <http://www.w3.org/TR/xmlschema-1/>
- [36] N. Klarlund, A. Moller, and M. I. Schwatzbach, “DSD: A Schema Language for XML”, ACM SIGSOFT Workshop on Formal Methods in Software Practice, Portland, OR, Aug. 2000
- [37] M. Murata. “RELAX (REGular LAnguage description for XML)”, Aug. 2000.
- [38] J. Clark, “TREGX - Tree Regular Expressions for XML”, 2001.
- [39] J. Clark, and M. Murata (Eds), “RELAX NG Tutorial”, OASIS Working Draft, Jun. 2001, <http://www.oasis-open.org/committees/relax-ng/tutorial.html>.
- [40] M. Murata, D. Lee, and M. Mani, “Taxonomy of XML Schema Languages using Formal Language Theory”, ACM transactions on the Internet, 660-704, Nov. 2005.
- [41] J. Cheng, W. Ng, “XQzip: Querying Compressed XML Using Structural Indexing”, Proc. of EDBT, 2004.
- [42] A. Arion, A. Bonifati, G. Costa, S. D’Águanno, I. Manolescu, A. Pugliese, “Efficient Query Evaluation over XML Compressed Data”, Proc. of EDBT , 2004

- [43] G. Leighton, J. Diamond, T. Muldner, “AXECHOP: A Grammar-based Compressor for XML”, Data Compression Conference (DCC 2005), Snowbird, Utah, Proceedings of IEEE, pp. 467, 2005.
- [44] J. Adiego, P. de la Fuente, and G. Navarro, “Merging prediction by partial matching with structural contexts model”, Data Compression Conference (DCC 2004), Snowbird, Utah, Proceedings of IEEE, pp. 522, 2004.
- [45] J.Cheney, “An empirical evaluation of simple DTD-conscious compression techniques”. Proceedings of the Eighth Workshop on the Web and Databases (WebDB),p.43–48,2005.
- [46] S. Hariharan, P. Shankar.S. “Evaluating the Role of Context in Syntax Directed Compression of XML Documents”. Data Compression Conference (DCC 2006), Snowbird, Utah, Proceedings of IEEE, pp. 453, 2006.
- [47] J.Cheney. “Tradeoffs in XML database compression”. Data Compression Conference (DCC 2006), Snowbird, Utah, Proceedings of IEEE, pp. 392–401, 2006.
- [48] L.Segoufin, V.Vianu, “Validating streaming XML documents”, Proceedings of Symposium on Principles of Database Systems (PODS) pp. 53 - 64 , 2002
- [49] A. Averbuch, S. Harrusi, A. Yehudai, XML parser, Patent Application #20060117307, 2005.
- [50] S. Harrusi, A. Averbuch, A. Yehudai, “XML Syntax Conscious Compression”, Data Compression Conference (DCC 2006), Snowbird, Utah, Proceedings of IEEE, pp. 402–411, 2006.
- [51] Y. G. Li, S. Bressan, G. Dobbie, Z. Lacroix , T.A Mong, L. Lee, U. Nambiar, “XOO7: applying OO7 benchmark to XML query processing tool”, Proceedings of international conference on Information and knowledge, pp. 167 - 174, 2001.
- [52] K. Runapongsa, J.M. Patel, H.V. Jagadish, Y. Chen, S. Al-Khalifa, “The Michigan benchmark: towards XML query performance diagnostics”, Informarion Systems 31(2), pp. 73-97. 2006.
- [53] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse. “XMark: A Benchmark for XML Data Management”. In Proceedings of VLDB, pp. 974-985 , 2002.
- [54] E.Rahm, T.Bohme, “XMach-1: A Multi-User Benchmark for XML Data Management”, Proceedings VLDB workshop Efficiency and Effectiveness of XML Tools and Techniques (EEXTT2002), 2002 (Invited Talk).
- [55] <http://sourceforge.net/projects/expat/>.