Using Hardcoded Hedge-Automata for Compressing Structured Documents

Aditya Nori Priti Shankar Helmut Seidl

IISc-CSA-TR-2002-1

http://archive.csa.iisc.ernet.in/TR/2002/1/

Computer Science and Automation
Indian Institute of Science, India

July 2002

Using Hardcoded Hedge Automata for Compressing Structured Documents

Aditya Nori* Priti Shankar† Helmut Seidl‡

Abstract

We propose a scheme for compressing tree structured data which uses a hardcoded pushdown hedge automata. Our algorithm is a lossless adaptive algorithm where the hardcoded model is used in conjunction with an arithmetic compressor to produce a compressed version of the tree. The structure of the model mirrors the syntactic specification of the tree structure. A quantitative analysis of the advantages that accrue from such a model is provided.

1 Introduction

Extensible Markup Language(XML) [14] is a standard meta language to describe a class of data objects, namely XML documents and specify how they are to be processed by computer programs. XML is rapidly becoming a standard for the creation and parsing of documents. However, a significant disadvantage is document size, which is a consequence of verbosity. It is commonly observed that non-standardized text formats for describing equivalent data are significantly shorter. Theoretically, therefore, one should be able to compress XML documents down to the same size as the compressed versions of their non-standard counterparts. XML documents have their structure specified by Document Type Definitions(DTD) which specify the syntax of the documents. From an information-theoretic standpoint, portions of the document that have to do with its layout should not add to its entropy. It is therefore natural to investigate the use of syntactic models for the compression of such data.

The use of syntax in the compression of program files is not new. Cameron [2] has used context free grammars for compressing programs. 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 context free grammar. 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 [5, 6, 7, 8]. Compression schemes for XML [3, 10] have tried to take advantage of the structure in XML data by either transforming the file after parsing, breaking up the tree into components or injecting hierarchical element structure symbols into a model that multiplexes several models based on XML's syntactic structure. The approach here makes crucial use of the syntax specified by the DTD of the document. The hard-coded model mirrors the DTD in the same way that recursive descent parsers mirror the underlying LL(1) grammar. What is of importance here is that the model tracks the structure of the document, and is able to make accurate predictions of the expected symbols. More importantly, whenever the predicted symbol is unique, there is no need to encode it at all as the decoder generates the same hardcoded model from the DTD and is thus able to generate the unique expected symbol. Most markup symbols fall into this category of symbols. Section 2.1 provides some background on

*Computer Science and Automation, Indian Institute of Science, Bangalore 560 012, India, aditya@csa.iisc.ernet.in
†Computer Science and Automation, Indian Institute of Science, Bangalore 560 012, India, priti@csa.iisc.ernet.in
‡Universitat Trier, FB IV - Informatik, D-54286 Trier, Germany, seidl@uni-trier.de

arithmetic coders. Section 3 describes the scheme, and Section 4 provides a quantitative analysis. In Section 5, we describe the architecture of an XML compressor-decompressor generator tool based on the theory developed in the earlier sections. Finally Section 6 concludes the paper.

2 Arithmetic Coding and Finite Context Modeling

2.1 Arithmetic Coding

Arithmetic coding does not replace every input symbol with a specific code. 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. In the example that follows(adapted from [13]), suppose the alphabet is $\{a, b, c, d, e, !\}$, and a fixed model is used which probabilities shown in the following table.

Symbol	Probability	Range
a	0.2	[0-0.2)
b	0.3	[0.2,0.5)
c	0.1	[0.5,0.6)
d	0.2	[0.6,0.8)
e	0.1	[0.8,0.9)
!	0.1	[0.9,1)

Suppose we wish to send the message *bacc!*. Initially, both the encoder and the decoder know that the range is [0, 1). After seeing the first symbol b the encoder narrows it down to [0.2,0.5)(this is the range that the model allocates to symbol b). The second symbol a, the interval is further narrowed to one-fifth of itself, since a has been allocated [0,0.2). Thus the new interval is [0.2,0.26). After seeing the first c the narrowed interval is [0.23,0.236), and after seeing the second c the new interval is [0.233,0.2336). Finally on seeing !, the interval is [0.23354,0.2336); knowing this to be the final range the decoder can immediately deduce that the first character was b. Now the decoder simulates the action of the encoder, since the decoder knows that the interval [0.2,0.5) belonged to b, the range is expanded to [0.2,0.26). Continuing this way, the decoder can completely decode the transmitted message. It is not really necessary for the decoder to know both ends of the range produced by the encoder. Instead, a single number in the range (say 0.23355 in our example) will suffice. However the decoder will face the problem of detecting the end of message, to determine when to stop decoding. To resolve the ambiguity, we ensure that each message ends with a special EOF symbol known to both encoder and decoder. From the discussion above, it should be clear, that for this scheme to be effective, the model should produce probabilities that deviate from a uniform distribution. The better the model is at making such predictions, the better the compression ratios will be.

2.1.1 Finite Context Modeling

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 decompressor 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. The decompressor similarly decodes a symbol using the existing model and then updates the model. Since there are potentially q^k possibilities for level k contexts where q is the size of the symbol space, updation can be a costly process. This causes arithmetic coding to be somewhat slower than dictionary based schemes like the Ziv-Lempel scheme [15].

3 Hardcoded Pushdown Hedge Automata for XML Documents

The scheme proposed in this paper assumes that the DTD describing the data is known to both the sender and the receiver. Each element of the DTD is actually a specification of a tree language. Typically, an element 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 ("processing instructions"). Opening element tags can have attributes with values. Each element definition can be viewed as a tree rooted at the element opening tag having a sequence of subtrees specified by the regular expression with possibly character data between consecutive trees. One commonly used characterization of such a sequence is a *hedge* [11]. Regular Hedge Grammars have been used as a mechanism for generating hedges and consequently as a formal representation of XML Schema. We borrow from Murata [11] the definition of a regular hedge grammar.

Definition 1 *A regular hedge grammar (RHG) G is a 5-tuple $< \Sigma, X, N, P, S >$ where:*

1. *Σ is a finite set of symbols,*
2. *X is a finite set of variables,*
3. *N is a finite set of non-terminals,*
4. *P is a set of production rules of one of the two forms below*
   - *$n \to x$ where $n \in N, x \in X$,*
   - *$n \to a < r >$ where $n \in N, a \in \Sigma$ and r is a regular expression over the set of non-terminals.*
5. *S is the starting non-terminal.*

Derivation sequences are defined in the usual way. However, we note that the objects being derived are trees. An X-derivation tree, D_X, where $X \in N$ for G has the following properties:
1. The root of the tree has label X.
2. If X is an internal node, then the subtree rooted at X is one of the following three types; (For describing trees we use the usual list notation)

- $X(D_Y)$ if $X \to Y$ is a production and D_Y is a derivation tree rooted at Y.
- $X(a)$ if $X \to a, a \in \Sigma$ is a production of P.
- $X(a(D_{X_1}, D_{X_2}, \ldots D_{X_k}))$ if $X \to a < r >$ is an element of P with $X_1 X_2 \ldots X_k$ an element of the language specified by the regular expression r.

Let $TREES(A)$ be the set of trees with symbols from Σ. The language defined by the grammar is then the set

$$L(G) = \{t \mid t \in TREES(A),\ and\ S \stackrel{*}{\Rightarrow} t\}$$

Example 1 *Consider a DTD defined as follows:*

```
<!DOCTYPE addressBook[
<!ELEMENT addressBook(card*)>
<!ELEMENT card((name|(givenName,familyName)),email, note?)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT email(#PCDATA)>
<!ELEMENT familyName(#PCDATA)>
<!ELEMENT note(#PCDATA)>
]>
```

Below is an instance of an XML document conforming to this DTD.

```
<addressBook>
<card>
<givenName>Aditya</givenName>
<familyName>Nori</familyName>
<email>aditya@csa.iisc.ernet.in</email>
</card>
<card>
<name>Priti Shankar</name>
<email>priti@csa.iisc.ernet.in</email>
<note>Aditya's advisor</note>
</card>
</addressBook>
```

A hedge grammar for this DTD is given below:

$G = (\Sigma, X, N, P, n_a)$

$\Sigma = \{\texttt{addressBook, card, name, givenName, familyName, email, note}\}$

$X = \{\texttt{\#PCDATA}\}$

$N = \{n_a, n_c, n_n, n_g, n_f, n_e, n_o, n_\#\}$

$$\begin{aligned} P = \{ & n_a \rightarrow \texttt{addressBook} < (n_c)^* >, \\ & n_c \rightarrow \texttt{card} < (n_n|(n_g, n_f)), n_e, n_o? >, \\ & n_n \rightarrow \texttt{name} < n_\# >, \\ & n_g \rightarrow \texttt{givenName} < n_\# >, \\ & n_f \rightarrow \texttt{familyName} < n_\# >, \\ & n_e \rightarrow \texttt{email} < n_\# >, \\ & n_o \rightarrow \texttt{note} < n_\# >, \\ & n_\# \rightarrow \texttt{\#PCDATA}\} \end{aligned}$$

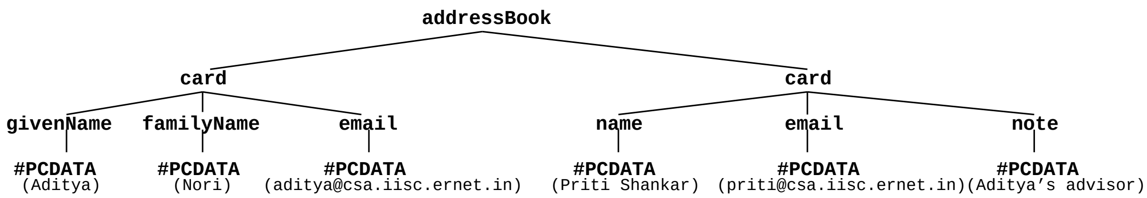

Figure 1: Hedge for the XML document in example 1

A pictorial representation of a hedge is given in Figure 1. Each node corresponds to an element and the text in parentheses represents the character data content of an element.

The DFA for the right hand side of the production for the non-terminal n_c *is shown in Figure 2.*

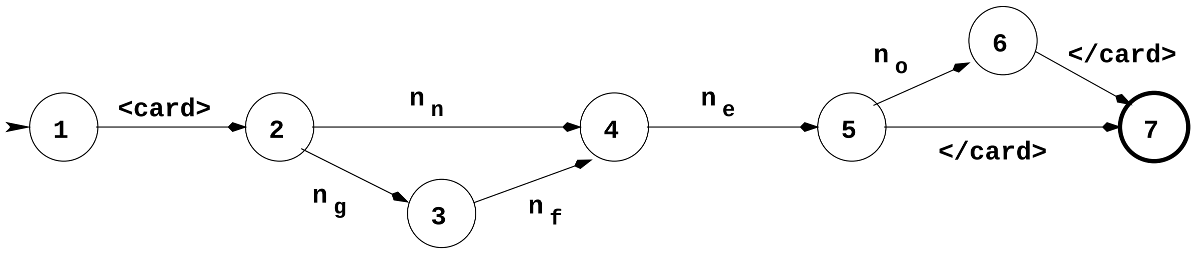

Figure 2: DFA for the rhs of the production for n_c in example 1

A parse tree for the XML document in the example is also shown in Figure 3.

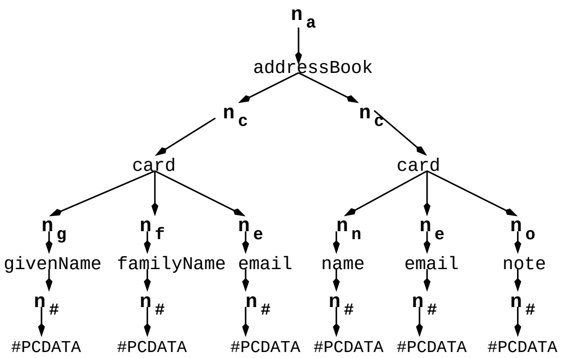

Figure 3: A parse tree for the XML fragment in example 1

We note that for each element there is a beginning and an end tag. Also the productions look very similar to those of an Extended Context Free syntax. The scheme we use is restricted to a class of hedge grammars satisfying a condition equivalent to the ELL(1) condition. The basic idea here is that we will have one procedure for each non-terminal and the body of the procedure will be structured based on the right-hand side of the unique production for that non-terminal.

A $*$ operator will be translated into a `while` loop, a ? operator into an `if-then-else` statement and alternates | will be implemented as cases of a `switch` statement. For parsing to proceed deterministically we have conditions that are identical to the ELL(1) conditions of the Extended Context Free Grammars. Basically this forces parsing decisions to be unambiguous at each stage.

The conditions for deterministic parsing of trees generated by a hedge grammar are practically identical to those for deterministic parsing of strings generated by an extended context free grammar [1]. For the sake of brevity, we do not repeat the conditions, but concentrate on the hardcoded models for compression and decompression. The first observation that is employed is the fact that the right-hand side of each production of the hedge grammar corresponds to a deterministic finite state automaton(DFA). For example, the production $n_c \rightarrow$ `card` $< (n_n|(n_g, n_f)), n_e, n_o? >$ corresponds to the DFA shown in Figure 2. 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. These states correspond to branch points in the program simulating the DFA. An arithmetic encoding procedure is called at each such state for each non-terminal. As we observed in Section 2.1, the arithmetic encoder maintains tables of frequencies which it updates each time it encodes a symbol. Here there will be a set of tables for each non-terminal, one for each branch point in the routine corresponding to the non-terminal. The symbols in the table will be the set labeling the outgoing transitions from that state. The decoder mimics the action of the encoder generating symbols that are certain and using the arithmetic decoder for symbols that are not.

We illustrate the encoder and decoder generated for the procedure for the non-terminal n_c. A call to $arith-encode(n, i, token)$ will access the frequency table associated with non-terminal n at the i^{th} branch point of the procedure associated with n, and increment the count associated with $token$ after encoding it (since the context is hardcoded, order-0 tables are maintained at each state). Let on_c and cn_c be the tokens corresponding to the opening and closing tags of non-terminal n_c. Below is the procedure generated for the encoding model for non-terminal n_c.

procedure p_{n_c}*– encode*
    if $next\text{–}token = on_c$
        $get\text{–}token$
    else $error$
    endif
    $arith-encode(n_c, 1, next\text{–}token)$
    case $next\text{–}token$ **of**
        $on_n : p_{n_n}\text{–} encode$
        $on_g : p_{n_g}\text{–} encode\ ;\ p_{n_f}\text{–} encode$
    esac
    $p_{n_e}\text{–} encode$
    $arith-encode(n\_c, 2, next\text{–}token)$
    if $next\text{–}token = on_o$
        $p_{n_o}\text{–} encode$
    if $next\text{–}token = cn_c$
        $get\text{–}token$
    else $error$
end procedure

The procedure generated for the decoding model is similar and is given below.

procedure p_{n_c}*– decode*
    generate on_c

$arith-decode(n_c, 1, next\text{–}token)$
case $next\text{–}token$ **of**
$on_n : p_{n_n}\!-decode$
$on_g : p_{n_g}\!-decode$; $p_{n_f}\!-decode$
esac
$p_{n_e}\!-decode$
$arith-decode(n_c, 2, next\text{–}token)$
if $next\text{–}token = on_o$
$p_{n_o}\!-decode$
generate cn_c
end procedure

There are a few observations to be made here. We note that the counts for characters that make up the non mark-up parts of the document are automatically stored in different frequency tables for each different instance of the variable #pcdata. Attribute names and values are also similarly handled.

4 Analysis

We refer to our model as the *Prediction by Condensed Future* (PCF) model, and compare it with the Prediction by Partial Match (PPM) arithmetic coding scheme. We now present a simple analysis to show that this model is at worst, as good as the PPM and at best can reduce the message length considerably. For a non-terminal $A \in (N \cup T)^*$, define a set

$$LAST_k(A) = \{z | z \in T^*, A \stackrel{*}{\Rightarrow} wz \ for \ some \ w \in (N \cup T)^* \ and \ length(z) = k\}$$

In other words, $LAST_k(A)$ is the set of all distinct suffixes of length k of strings of input symbols that can be derived from A. Let $\{\beta_i \in (N \cup T)^*\}_{1 \le i \le l}$ be distinct suffixes of left sentential forms such that

$$S \stackrel{*}{\Rightarrow} w_1 A \beta_1, \ S \stackrel{*}{\Rightarrow} w_2 A \beta_2, \ \ldots, S \stackrel{*}{\Rightarrow} w_l A \beta_l$$

and $z \in LAST_k(A)$. Clearly, an order k PPM model would be in a single state (say q_z) for each distinct $z \in LAST_k(A)$. The PCF model, on the other hand would be in different states for each $\{\beta_i\}_{1 \le i \le l}$ (the states here refer to the distinct stack configurations of the pushdown automaton). Let $q_1, q_2, \ldots, q_l$ be the distinct states of the PCF model which correspond to stack contents $\{\beta_i\}_{1 \le i \le l}$. We first show that in the worst case, the average entropy H_{avg} over this set of states is bounded by the entropy H_{q_z} at state q_z. Let q_z have outgoing transitions on symbols $\{a_1, a_2, \ldots, a_n\}$. Let f_q denote the total frequency of symbols at a state q. f_{iq} denotes the frequency of symbol a_i at state q. Denote f_{q_j} by f_j, and f_{iq_j} by f_{ij}, $1 \le i \le n, 1 \le j \le l$. The following equations relate the counts at the states q_i, $1 \le i \le l$ (in PCF), to those at state q_z (in PPM).

$$\sum_{j=1}^{l} f_j = f_{q_z} \ ; \ \sum_{j=1}^{l} f_{ij} = f_{iq_z}$$

Since the probabilities of symbols are approximated by frequency ratios in the adaptive model, we compute entropies in terms of these ratios.

The entropy for state q_z is defined as follows

$$H_{q_z} = \sum_{i=1}^{n} \frac{f_{iq_z}}{f_{q_z}} \log \frac{f_{q_z}}{f_{iq_z}}$$

Now given that the PPM model is in state q_z, the PCF model could be in any one of the states $q_1, q_2, \ldots q_l$. Let p_j denote the conditional probability that PCF is in state $q_j, 1 \le j \le l$, given that PPM is in state q_z.

$$p_j = \frac{f_j}{f_{q_z}}$$

The average entropy H_{avg} over all the states $\{q_1, \ldots, q_l\}$ is given by

$$H_{avg} = \sum_{j=1}^{l} p_j H_{q_j} = \sum_{j=1}^{l} p_j \sum_{i=1}^{n} \frac{f_{ij}}{f_j} \log \frac{f_j}{f_{ij}}$$

Interchanging the summations, we have

$$\sum_{i=1}^{n} \sum_{j=1}^{l} \frac{f_{ij}}{f_{q_z}} \log \frac{f_j}{f_{ij}}$$

Define

$$h(a_i) = \sum_{j=1}^{l} \frac{f_{ij}}{f_{q_z}} \log \frac{f_j}{f_{ij}} \ ; \ h_{q_z}(a_i) = \frac{f_{iq_z}}{f_{q_z}} \log \frac{f_{q_z}}{f_{iq_z}}, \ 1 \le i \le n$$

In order to prove that $H_{avg} \le H_{q_z}$, it suffices to show that $h(a_i)$ is at most $h_{q_z}(a_i)$ for each i, $1 \le i \le n$. Define $\Delta_i = h(a_i) - h_{q_z}(a_i)$. Therefore, we have

$$\Delta_i = \sum_{j=1}^{l} \frac{f_{ij}}{f_{q_z}} \log \frac{f_j}{f_{ij}} - \frac{f_{iq_z}}{f_{q_z}} \log \frac{f_{q_z}}{f_{iq_z}}$$

Replacing f_{iq_z} by $\sum_{j=1}^{l} f_{ij}$

$$\Delta_i = \sum_{j=1}^{l} \frac{f_{ij}}{f_{q_z}} \log \frac{f_j}{f_{ij}} - \frac{\sum_{j=1}^{l} f_{ij}}{f_{q_z}} \log \frac{f_{q_z}}{f_{iq_z}}$$

$$= \sum_{j=1}^{l} \frac{f_{ij}}{f_{q_z}} \log \frac{f_j f_{iq_z}}{f_{ij} f_{q_z}}$$

Using Jensen's inequality

$$\Delta_i \le \log \left[\sum_{j=1}^{l} \frac{f_j f_{iq_z}}{f_{q_z}^2} \right]$$

But $\sum_{j=1}^{l} \frac{f_j f_{iq_z}}{f_{q_z}^2} = \frac{f_{iq_z}}{f_{q_z}}$, which means

$$\Delta_i \le \log \frac{f_{iq_z}}{f_{q_z}}$$

We also know that $f_{iq_z} \le f_{q_z}$. Therefore

$$\Delta_i \le 0 \Rightarrow h(a_i) \le h_{q_z}(a_i)$$

Thus it follows that $H_{avg} \le H_{q_z}$.

In the best case, we have a partition of the alphabet set $\{a_1, a_2, \ldots, a_n\}$ into l blocks such that each block contains exactly one alphabet. Thus, $l = n$ and the elements of f_{ij} are either 0 or f_{iq_z}, $1 \le i \le n$. Also, the f_j are a permutation of the f_{iq_z}. In such a case $H_{avg} = \sum_{j=1}^{l} \frac{f_j}{f_{q_z}} \sum_{i=1}^{l} \frac{f_{ij}}{f_j} \log \frac{f_j}{f_{ij}}$.

Since $f_{ij} = 0$ or one of f_j, we have $H_{avg} = 0$. Thus the net contribution of these states to the average entropy is zero.

The previous analysis indicates the advantages of "context refinement", so to speak, where a single state of the PPM model is refined into a set of states of the PCF model. In some cases, the opposite, i.e. "context merging" is beneficial in terms of reducing the number of tables to be maintained. We explain the basis for merging as follows.

Kuich [9] has defined the entropy of a context-free language as follows.

Definition 2 *Let L be a language and let $u(n)$ be the* structure function *of L, that is, $u(n)$ is the number of distinct words of length n contained in L. Then the function $f(z)$ of the complex variable z*

$$f(z) = \sum_{n=1}^{\infty} u(z) z^n$$

is called the structure generating function *of L. The above power series converges absolutely in the disk $|z| < R$, where R is the radius of convergence and is given by the Cauchy-Hadamard theorem as*

$$R = \frac{1}{\Lambda}, \text{ where } \Lambda = \limsup_{n\to\infty} \sqrt[n]{u(n)}$$

The entropy *H associated with a language L (and hence, with the grammar generating L) is defined by*

$$H = \log \Lambda \tag{1}$$

We can extend this notion to the entropy of suffixes of left sentential forms of the grammar. Define the set of *valid continuations* of any prefix wA, $w \in T^*$, $A \in N$ of a left sentential form as the set

$$VC(wA) = \{z \mid wA\beta \text{ is a left sentential form and } \beta \stackrel{*}{\Rightarrow} z\}$$

We can thus associate with each state q of the PCF model, a set of strings. This set of strings is associated with a condensed future (i.e., the contents on the pushdown stack). The entropy at state q is given as in equation 1, where $u(n) = |\{z \in VC(wA) | \; length(z) = n\}|$, that is, the number of distinct strings of length n which are valid continuations of strings derived from wA.

Consider $A \in N$, and let $z_1, z_2, \ldots, z_k$ be distinct strings in $LAST_k(A)$. There exists a prefix wA of a left sentential form such that $wA \stackrel{*}{\Rightarrow} ww_iz_i,\ 1 \le i \le k$. Since the z_i are all distinct, they lead to different states in the order-k PPM model. However, since each of the strings ww_iz_i has an identical set of *futures* (represented by the contents of the stack), they need not be distinguished in terms of the state of the model. Indeed, they are not in the PCF model.

5 Design of an XML Compressor/Decompressor Generator

We will now describe the design rationale for an XML compressor generator based on the theory developed in the earlier sections. Section 3 described procedures (p_{n_c}*– encode*, p_{n_c}*– decode*) derived from a hedge grammar which performed the task of parsing a hedge which conforms to the grammar in question. In addition to parsing, these procedures were also used for the purposes of compression and decompression. Therefore, given a DTD (which is a hedge grammar describing the structure of an XML document), it should be possible to automatically generate procedures for compression/decompression in a similar manner. Figure 4 shows the architecture of the XML compressor system. We will now describe the components of this architecture and also state briefly how each of these may be implemented.

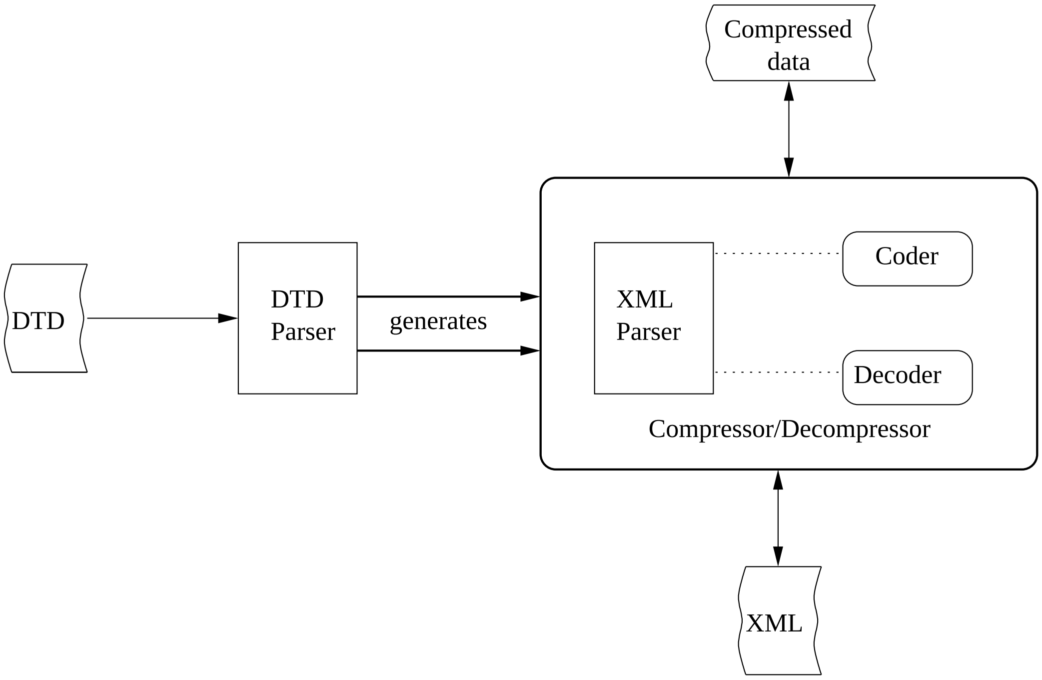

Figure 4: Architecture of the Compressor/Decompressor Generator

1. **DTD Parser and XML Parser**. DTD Parser takes as input a DTD $\mathcal{D}$ and generates a recursive descent style parser $\mathcal{XP}$ for XML documents conforming to $\mathcal{D}$. $\mathcal{XP}$ consists of procedures (for example, $p_{n_c}\!-encode$) for each non-terminal in the grammar described by $\mathcal{D}$. Calls to the arithmetic coder/decoder are made at appropriate points in $\mathcal{XP}$. The DTD Parser is a bottom up LR parser generated by the parser generator tool *yacc* which basically performs a syntax directed translation over the input DTD $\mathcal{D}$ to generate $\mathcal{XP}$.

2. **Coder and Decoder**. Coder and Decoder are respectively the arithmetic compressor and decompressor routines as described in [12]. These are modified such that the input buffer from which data (that is to be compressed/decompressed) is read, is now a parameter to the coding (decoding) function. This change is necessary in order to handle encoding of PCDATA and attribute values occurring in different contexts.

Therefore, the compressor-decompressor system works in the following manner. The DTD Parser is given a DTD $\mathcal{D}$ as input which generates the compressor-decompressor system (call this CDS). CDS when given an XML document $\mathcal{X}$ which respects $\mathcal{D}$ as input, compresses $\mathcal{X}$, else reports a syntax error. On the other hand, when CDS is provided a compressed XML document (whose uncompressed version conforms to the DTD $\mathcal{D}$), inflates it to restore the original XML document.

6 Conclusion

We have presented a scheme for the compression of tree structured documents using arithmetic coding where the underlying model is a pushdown hedge automaton. The advantage of using such a model is that accurate predictions of probabilities of symbols are possible. Further this technique provides a systematic approach to the automatic generation of hardcoded compressor models from

DTD specifications. Future work will concentrate on implementation of the techniques described here and on schemes for querying compressed documents.

References

[1] R. C. Backhouse. Syntax of Programming Languages - Theory and Practice, *Prentice Hall International, London*, 1979.

[2] Robert D.Cameron. Source Encoding Using Syntactic Information Source Models. *IEEE Trans. on Inform. Theory*, 34(4):843-850, 1988.

[3] James Cheney. Compressing XML with Multiplexed Hierarchical Models. *Proceedings of the 2001 IEEE Data Compression Conference*, pp. 163-172.

[4] John G. Cleary and I. H. Witten. Data Compression using Adaptive Coding and Partial String Matching. *IEEE Trans. Comm.*, 32(4), 396-402, Apr 1984.

[5] Jens Ernst, William Evans, Christopher W.Fraser, Steven Lucco and Todd A. Proebsting. Code compression. *PLDI '97*:358-365, 1997.

[6] M. Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems. *TR 97-04, Department of Information and Computer Science, University of California, Irvine*, 1997.

[7] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87-94, December 1997.

[8] C.W. Fraser. Automatic Inference of Models for Statistical Code Compression, *PLDI '99*:242-246, 1999.

[9] W. Kuich. On the Entropy of Context-free Languages. *Information and Control*, 16: 173-200, 1970.

[10] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data, *Proceedings of ACM SIGMOD*, 2000.

[11] Murata Makoto. Hedge Automata: a formal model for XML schemata. `http://www.oasis-open.org/cover/topics.html#forestAutomata`.

[12] Mark Nelson. Arithmetic Coding. *Dr. Dobbs Journal* 1991.

[13] Ian H. Witten, Radford M.Neal and John G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6): 520-540, June 1987.

[14] Extensible Markup Language (XML) 1.0. W3C Recommendation 10 Feb, 1998. Reference: REC-xml-19980210. `http://www.w3.org/TR/REC-xml`.

[15] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. on Inform. Theory*, IT-23(3), 1977.