Schema-Based Compression of XML Data with Relax NG

Christopher League and Kenjone Eng
Long Island University Computer Science, Brooklyn, NY, USA
Email: {christopher.league, kenjone.eng}@liu.edu

Abstract— **The extensible markup language XML has become indispensable in many areas, but a significant disadvantage is its size: tagging a set of data increases the space needed to store it, the bandwidth needed to transmit it, and the time needed to parse it. We present a new compression technique based on the document type, expressed as a Relax NG schema. Assuming the sender and receiver agree in advance on the document type, conforming documents can be transmitted extremely compactly. On several data sets with high tag density this technique compresses better than other known XML-aware compressors, including those that consider the document type.**

***Index terms*—XML, data compression, tree compression, Relax NG, compact binary formats**

I. Motivation

In recent years, the extensible markup language XML [2] has become indispensable for web services, document markup, conduits between databases, application data formats, and in many other areas. Unfortunately, a significant disadvantage in some domains is that XML is extremely verbose. Although disk capacity is less often a concern these days, transmitting XML-tagged data still requires significantly more bandwidth and longer parse times (compared to a custom binary format, for example).

Compression of XML has been studied from a variety of perspectives. Some researchers aim to achieve minimal size [3], [4], [5], others focus on efficient streaming [6], [7], [8] – a balance between bandwidth and encode/decode times – and still others answer XML queries directly from compressed representations [9]. Representations that support queries are necessarily larger than those that do not; Ferragina et al. [10] report increases of 25 to 96% compared to opaque representations. This is not all that bad if querying is a requirement, but we assume it is not and aim for the smallest possible size. Following Levene and Wood [11], we study how a schema (document type) can be used to reach that goal.

Traditionally, one writes a document type definition (DTD) to constrain the sequencing of tags and attributes in an XML document. For most commonly-used XML formats, a DTD already exists. The DTD language is simple, but not terribly expressive, so a few competitors have arisen. We focus in particular on *Relax NG* by Clark and Murata [12]. It is expressive, but unlike *XML Schema* [13] it has a clean formal model [14] that is the foundation of our compression technique.

```
start = stm

stm = element seq { stm+ }
    | element assign { var, exp }
    | element print { exp* }

exp = element num
        { attribute val {text}, empty }
    | element id
        { var, empty }
    | element binop
        { attribute id {text}?,
          op, exp, exp }
    | element exp
        { attribute ref {text} }

var = attribute var {text}

op  = attribute op
        { "add" | "sub" | "mul" | "div" }
```

Figure 1. Relax NG schema in compact syntax. It defines the grammar for a simple sequential language – after Wang et al. [15] – expressed in XML.

Figure 1 contains a simple Relax NG schema written using the compact syntax. (There is an equivalent XML syntax that would make this example about 4 times longer.) This schema specifies the grammar for a small sequential language with variables, assignment, numbers, and arithmetic expressions. Note the use of regular expression operators (`+*?|`) and the `id`/`ref` attributes for reusing common sub-expressions.

Our original motivation for studying XML compression was to investigate the use of XML to represent abstract syntax trees and intermediate languages in compilers. The language in figure 1 is based on an example given by Wang et al. [15]. They developed an abstract syntax description language (ASDL) for specifying the grammars of languages, generating type definitions in multiple languages, and automatically marshaling data structures to an

This paper is based on "Type-Based Compression of XML Data," by C. League and K. Eng, which appeared in the Proceedings of the IEEE Data Compression Conference (DCC), Snowbird, Utah, March 2007 [1].

```
<seq>
  <assign var='x'>
    <num val='7'/>
  </assign>
  <print>
    <binop op='add'>
      <binop id='r1' op='mul'>
        <id var='x'/>
        <id var='x'/>
      </binop>
      <binop op='sub'>
        <exp ref='r1'/>
        <num val='5'/>
      </binop>
    </binop>
  </print>
</seq>
```

Figure 2. XML code representing a program in the sequential language of figure 1. When run, the program would output 93 (= 49 + 49 – 5).

opaque but portable binary format. We hypothesize that XML and all its related libraries, languages, and tools could replace ASDL and prove useful elsewhere in the implementation of programming languages. Brabrand et al. [16] support this vision; they provide a tool for managing dual syntax for languages – one in XML, and one 'human-readable', much like the distinction Relax NG itself employs between its XML and compact syntax.

Figure 2 contains a small program in the sequential language, valid with respect to the Relax NG schema in figure 1. In conventional notation, it might be written as `x := 7; print [x*x + (x*x - 5)]`, but the common sub-expression `x*x` is explicitly shared via the `id`/`ref` mechanism. (One of the limitations of ASDL is that it could represent trees only, not graphs; however, managing sharing is essential for the efficient implementation of sophisticated intermediate languages [17].) These figures will serve as a running example in the remainder of the paper.

The main contribution of this paper is to describe and analyze a new technique for compressing XML documents that are *known* to conform to a given Relax NG schema. As a simple example of what can be done with such knowledge, the schema of figure 1 *requires* that a conforming document begins with either `<seq>`, `<assign>`, or `<print>`; so we need at most *two bits* to tell us which it is. Similarly, the `<binop>` tag has an optional `id` attribute and a required `op` attribute. Here, we need just one bit to indicate the presence or absence of the `id` and two more bits to specify the arithmetic operator. (The data values are separated from the tree structure and compressed separately.) For this system to work, the sender and receiver must agree in advance on precisely the same schema. In that sense, the schema is like a shared key for encryption and decryption.

We are not the first to propose compressing XML relative to the document type – an analysis of related work appears in section II – but ours is one of the few successful implementations, and the first such effort in the context of Relax NG. In section IV we report results that, on several data sets, improve significantly on other known techniques. The algorithm itself is detailed in section III and we close in section V by discussing limitations, consequences, and directions for future research.

II. Related work

There are a few common themes among XML compression techniques. One is separating the tree structure from the text or data. Another is finding ways to regroup the data elements for better performance.

There are also a few distinguishing features. Algorithms differ in whether they preserve the 'ignorable' white space outside of text nodes. (Those that discard it are not lossless in the traditional sense, but are perfectly acceptable given the content model of XML.) Also, algorithms differ in the extent to which they use the schema or DTD. Some ignore it entirely, others optionally use it to optimize their operations, and some (like us) regard it as essential.

A. General XML Compression

Liefke and Suciu [3] implemented XMill, one of the earliest XML-aware compressors. Its primary innovation was to group related data items into containers that are compressed separately. To cite their example, "all `<name>` data items form one container, while all `<phone>` data items form a second container." This way, a general-purpose compressor such as deflate [18] will find redundancy more easily. Moreover, custom semantic compressors may be applied to separate containers: date attributes, for example, could be converted from character data into a more compact representation. One of the main disadvantages of this approach is that it requires custom tuning for each data set to do really well.

Girardot and Sundaresan [6] describe Millau, a modest extension of WAP binary XML format, that uses byte codes to represent XML tag names, attribute names, and some attribute values. Text nodes are compressed with a deflate/zlib algorithm and the types of certain attribute values (integers, dates, etc.) are inferred and represented more compactly than as character data. Millau does not require a DTD, but if present it can be used to build and optimize the token dictionaries in advance.

Cheney's `xmlppm` [4] is an adaptation of the general-purpose *prediction by partial match* compression [19] to XML documents. The element names along the path from root to leaf serve as a context for compressing the text nodes. They provide benefits similar to those of XMill's containers. Adiego et al. [5] augment this technique with a heuristic to combine certain context models; this yields better results on some large data sets. In our experiments, the performance of `xmlppm` was nearly always competitive.; it was one of the main contenders.

Toman [20] describes a compressor that dynamically infers a custom grammar for each document, rather than using a given DTD. There is a surface similarity with our technique in that a finite state automaton models the element structure, and is used to predict future transitions. His system was tested on three classes of data, but it never beat xmlppm on either the compression ratio or the run time.

Skibiński et al. [7], [21] implemented the XML Word Replacing Transform, which we test in section IV. The *words* found in XML documents – including of course the element and attribute names themselves – can be extremely repetitive. xml-wrt replaces the most frequent words with well-encoded references to a dynamically-constructed dictionary. It also encodes any numeric data on a separate stream, using base 256. The tool is consistently competitive with xmlppm, but performs dramatically better on certain large data sets.

Chernik et al. [22] focus similarly on *syllables,* as a midpoint between character-based and word-based compression that is particularly effective for text in morphologically-rich languages such as German and Czech [23]. Their adaptation of syllable-based compression to XML approaches the effectiveness of XMill on text documents, and – interestingly – exceeds it slightly on Shakespeare.

B. Schema-based compression

We began this work with the idea that making effective use of the document type could significantly reduce the size of compressed representations. Others have explored this area in various ways, though not all have been successful.

Levene and Wood [11] proposed a DTD-based encoding in much the same spirit as our work; however, it is a theoretical result. There is no implementation or experimental data. Instead, they prove optimality, but under the assumption that the DTD is non-recursive, which is not generally true for our target application area (representing intermediate languages).

Sundaresan and Moussa [8] built on the work of Girardot (Millau), proposing *differential DTD compression.* Again, this sounds similar in spirit to our idea: do not transmit information that is already known from the DTD. Unfortunately, their paper is short on implementation details, and they report poor run-time performance (of that particular technique) on all but the simplest sorts of schemata. Specifically, it was unable to compress Hamlet[1] in a reasonable amount of time; they were forced to abort the computation and omit that test case from the results. (Hamlet is one of our benchmarks in section IV.)

Based on the effectiveness of xmlppm, we expected Cheney's DTD-conscious compress technique [24] to fare especially well, but as he reports, "for large data sets, dtdppm does not compress significantly better than xmlppm." In fact, in our tests, dtdppm never beat xmlppm by more than a few percent, and usually did worse.

More recent efforts are beginning to realize the potential of schema-based compression, and surpass xmlppm and xml-wrt on some data sets. Subramanian and Shankar [25] analyze DTDs to generate finite state automata and invoke arithmetic encoding at choice points. This technique is similar to ours, modulo the different schema languages, and their performance profile comes closest to ours. We will have more to say about their tool, xaust, in section IV.

Harrusi et al. [26] describe a staged approach, where a DTD (which they call a dictionary) is converted to a context-free grammar for a specialized parser. Although we do not have access to their implementation, a recent update [27] shows performance similar to xmlppm on standard benchmarks. The exception is the XOO7 benchmark [28], on which Harrusi et al. perform extremely well. We did not test against XOO7.

To summarize, XML-aware compression continues to be a vital research area, but empirical results for schema-based compression have so far been mixed.

III. The technique

Our technique has been implemented in Java, as a tool called rngzip that supports much of the same command-line functionality as gzip. See figure 4 for its help text; we describe the encoding and compression options later in this section.

We benefited greatly by using the Bali system by Kawaguchi,[2] a Relax NG validator and *validatelet compiler.* It builds a deterministic *tree automaton* [29], [14] from the specified schema. If also given an XML document, it checks whether the XML is accepted by the automaton. Alternatively, it can output a hard-coded validator for that particular schema in Java, C++, or C#.

We borrow the schema-parsing and automaton-building code from Bali. A diagram of the automaton induced from our sequential language schema is found in figure 3. The octagon-shaped states (1, 3, 8) are final/accept states. The transitions are labeled with XML tags or attribute predicates. The meanings of these predicates are described by example in the following table:

@id	contains an id attribute
!@id	does *not* contain an id attribute
!@~id	no attributes *except* possibly id
!@~(op|id)	no attributes except possibly op or id
!@*	no attributes at all

When the label of a transition is followed by a number, we jump to that state *as a subroutine* to validate the children of the current XML node (or the value of the current attribute).

Let's study how the program `<print><num val="42"/></print>` is validated by this automaton. Starting from state zero, there is a transition for `<print>`, so we push the target state 3 onto a stack and jump to state 2 as indicated by the transition. State 2 requires that the `<print>` just seen has no attributes. State 1 has a transition matching `<num>` so we push the target state 1 onto the stack and jump to 19. It requires that `<num>`

[1]from the Shakespeare XML corpus: http://www.ibiblio.org/bosak/xml/eg/shaks200.zip

[2]http://www.kohsuke.org/relaxng/bali/doc/

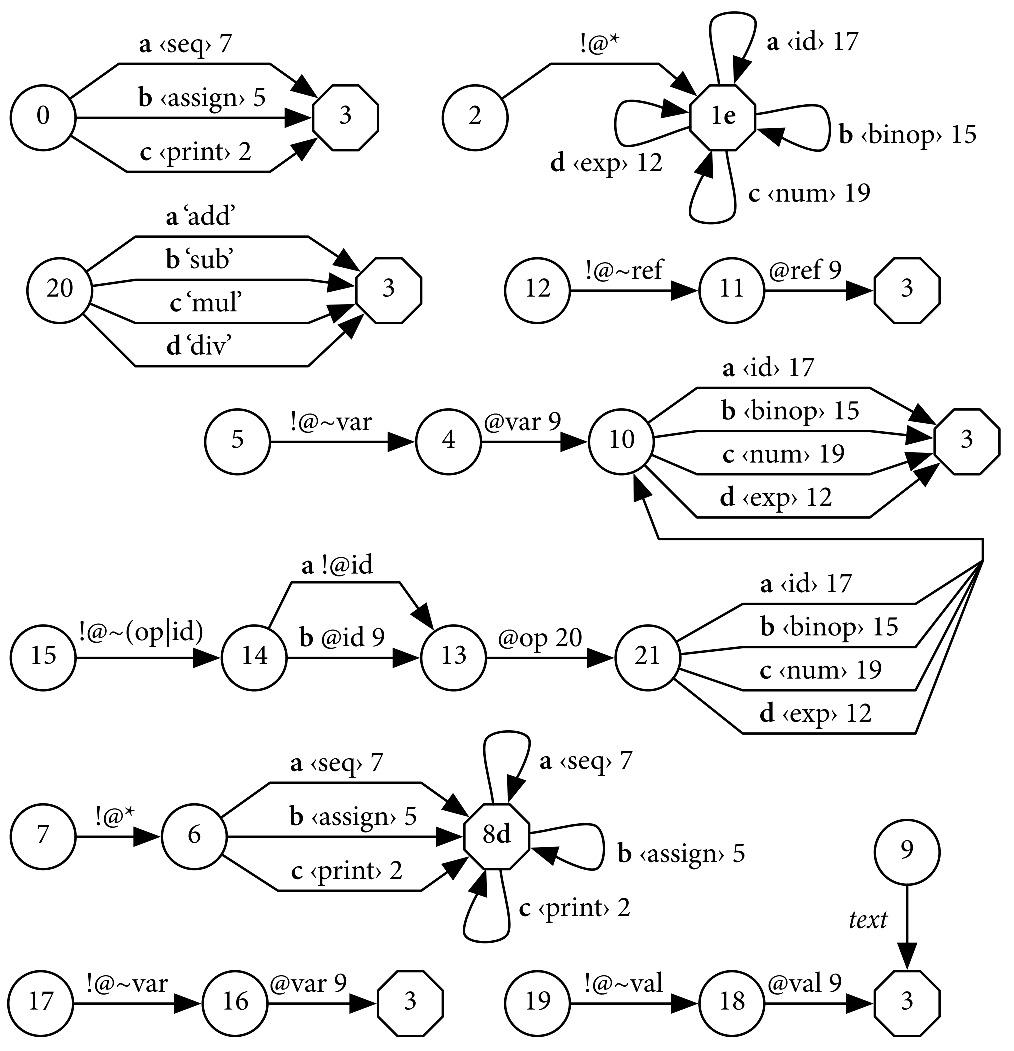

Figure 3. The tree automaton induced from the Relax NG schema in figure 1, and annotated with letters (a, b, c, . . .) at the choice points.

has no attributes except `val`, and that `val` contains *text.* We pop back up to state 1. The close tag `</print>` is accepted in this final state, and we pop back up to 3. The end of the document is accepted in this final state, and (since the stack is now empty) the document is declared valid.

Given this automaton, a receiver can reconstruct an entire XML document by transmitting very little information. Whenever there is a *choice point* in the automaton, we just transmit *which* transition was taken. Whenever we encounter the *text* transition, we transmit the matching text. Our program assigns unique labels to each outgoing transition from a choice point, shown in figure 3 as **a, b, c,** etc. The choice points in this automaton are states 0, 1, 6, 8, 10, 14, 20, and 21. Note that final states induce an additional choice: the automaton can either follow an outgoing transition, or stay and *accept*. This is why states 1 and 8 have a label *inside* the state.

So, in the case of the simple print statement above, we would just need to transmit **c, c,** `"42"`, **e**. Similarly, if we transmit **b,** `"x"`, **c,** `"8"`, the receiver ought to be able to reconstruct `<assign var="x"><num val="8"/></assign>`. Of course, it is even better than it looks: we are not transmitting the ASCII character **b;** since there are just 3 choices from state 0, we need only 2 bits to indicate which one, and 2 more bits for transition **c** from state 10. This is why it is critical that sender and receiver agree on precisely the same schema, and moreover that their implementations label transitions in the same order. (We had to impose a specific ordering of transitions on Bali, since it used a map based on object hash codes, which could vary between runs.)

Any white space outside of text nodes will be lost with this technique, but this is considered *ignorable* by the XML document model anyway, and it is trivial for the receiver to reformat the output if desired. In many cases, the decompressor will simply provide SAX events to some application, so the white space will not be missed; this is significantly more efficient than decompressing to a temporary file and then re-parsing it.

Now we step back to review other implementation details. Logically, the compressor outputs 3 distinct streams. The first is simply for configuration information: the URI of the schema, a checksum of the automaton, and

```
Usage: rngzip [options] [file ...]
Options:
 -c --stdout               write to standard output; do not touch files
 -D --debug                trace compressor; replaces normal output
 -E --tree-encoder=CODER   use method CODER for encoding the XML tree
 -f --force                force overwrite of output file
    --ignore-checksum      decompress even if schema changed (not recommended)
 -k --keep                 do not remove input files
 -p --pretty-print[=TAB]   line-break and indent decompressed output [2]
 -q --quiet                suppress all warnings
 -s --schema=FILE|URL      use this schema (required to compress)
 -S --suffix=.SUF          use suffix .SUF on compressed files [.rnz]
 -t --timings              output timings (implies -v)
 -T --tree-compressor=CM   compress the encoded XML tree using CM
 -v --verbose              report statistics about processed files
 -Z --data-compressor=CM   compress the data stream using CM

Modes:                     compress is the default; this requires -s
 -d --decompress           decompress instead of compress
 -i --identify             print information about compressed files
 -h --help                 provide this help
 -V --version              display version number, copyright, and license
    --exact-version        output complete darcs patch context

Coders: fixed huffman *byte
Compressors: none gz lzma bz2 *ppm
```

Figure 4. `rngzip` command line options.

an inventory of the other streams and how they are compressed. The second stream is for the bits that encode the tree structure, and the third is for the data values.

Physically, these streams are merged into one by dumping each buffer as it becomes full, with a short header encoding the stream ID and the buffer size. We call this a *multiplexed* stream and aimed initially to optimize it for efficient streaming, so that the memory requirements of both sender and receiver could be sub-linear in the total size of the document. Unfortunately, a particular property of Relax NG may conflict with the goal of streaming; see section V.

Our stream architecture permits three main points of customization, represented by the command-line options `-E`, `-T`, and `-Z` in figure 4.

`-E fixed|huffman|byte` – specifies how to encode the path taken at each choice point. Above, we referred to a choice point requiring two bits to distinguish between paths labeled a, b, or c. If we use $\lceil \log_2 n \rceil$ bits for every choice, and pack bits for consecutive choices together without regard for byte boundaries, that is the `fixed` encoding. We also provide an adaptive Huffman encoding [30] that eventually encodes more frequent transitions with proportionally shorter bit strings. Finally, we provide a byte-encoding; this is the same as fixed except that each choice occupies a full byte. It is meant to help a general-purpose compressor to find patterns in the bit stream.

`-T none|gz|lzma|bz2|ppm` – specifies what kind of compression to apply to the stream representing the tree structure. gz refers to the `GZIPOutputStream` that is part of the Java API; `lzma` is the Lempel-Ziv Markov chain Algorithm by Igor Pavlov;[3] `bz2` is the bzip2 algorithm by Julian Seward (Java implementation by Keiron Liddle);[4] `ppm` is Prediction by Partial Match [19] implemented by Bob Carpenter.[5]

`-Z none|gz|lzma|bz2|ppm` – specifies what kind of compression to apply to the stream containing the data (text) elements. For many data sets, this is the most significant setting. An earlier version of our implementation [1] supported gz only, and our performance on text-oriented documents such as Hamlet degraded to that of gzip.

The ppm setting, when specified for the data compressor (`-Z`), seeds the model with a checksum of the element names from the root to the text node or attribute value. This is an attempt to emulate the advantages of `xmlppm` within our tool. We ran all our benchmarks using every combination of these options, and selected one setting that represented the best compromise across all the tests. The outcome is explained in the next section.

IV. Experiments

We tested `rngzip` against several other compressors – generic, XML-aware, and schema-based – on a variety of data sets; the results are in figure 5. The competition included:

[3] `http://www.7-zip.org/sdk.html`
[4] `http://jakarta.apache.org/commons/sandbox/compress/`
[5] `http://www.colloquial.com/ArithmeticCoding/`

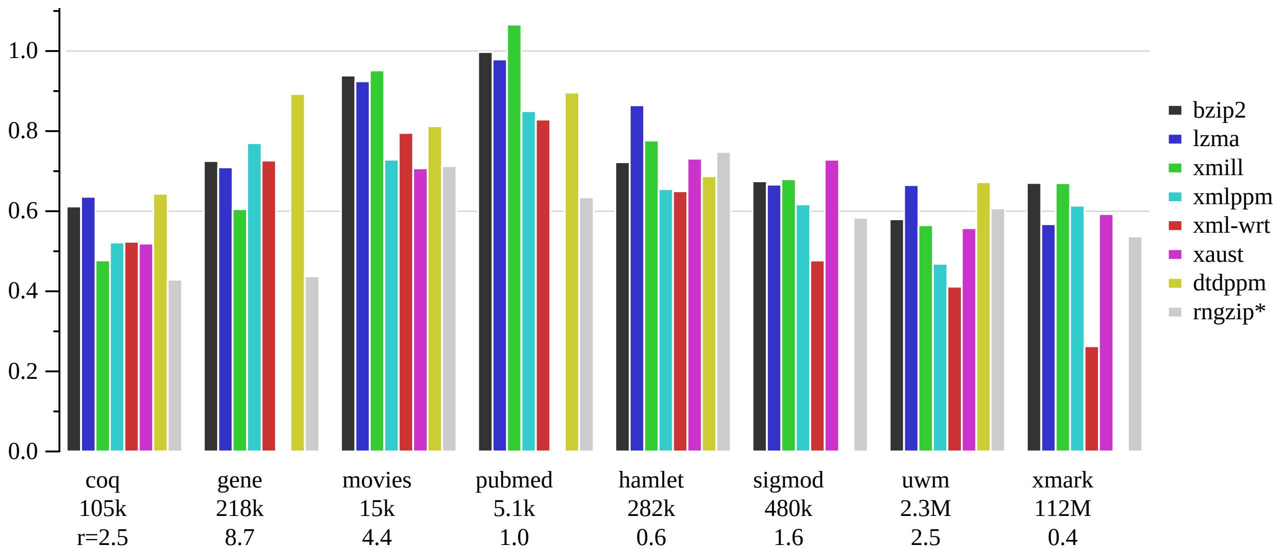

Figure 5. Effectiveness of several compressors over various data sets. The y-axis depicts compressed size, relative to gzip at 1.0. These are averages over many document instances, so error bars show one standard deviation above and below the mean. Beneath the data set labels are the average uncompressed sizes, the number of documents in the set (n), and the ratio of tags to text in the original (r, excluding ignorable spaces between tags).

- three generic compressors: `gzip` 1.3.5 by Jean-loup Gailly, `bzip2` 1.0.3 by Julian Seward, and `lzma` 4.32 by Igor Pavlov;
- three XML-aware implementations: XMill 0.8 [3], `xmlppm` 0.98 [4], and `xml-wrt` 3.0 [21];
- two schema-based systems: `xaust` [25] and `dtdppm` 0.5 [24].

We ran all of them without any options, except for `xml-wrt`, where we used `-l11` (PAQ compression). We did not specify any custom containers for XMill; they would need to be tuned separately for each schema.

Additionally, we tested 60 different combinations of options for `rngzip`, but for the sake of fairness, we chose just one setting (specified later) to use for all the results reported here. On some benchmarks, custom settings could have improved our performance by a few percent.

We reused some of the data sets from our previous work [1] in order to illustrate the improvement, but we now report on a single representative instance of each schema, so as not to muddy the results with averages and error bars. We also added two new data sets: Coq, because it is closest to our target application area; and XMark because it facilitates comparisons with several other systems that we could not test directly. Here are details about each benchmark:

- *coq* is an explicit representation of a proof about set membership[6] from the Coq proof assistant. Asperti et al. [31] advocate using XML for managing mathematical knowledge generally, and the XML export of Coq proofs is a result of their project. Since Coq is based on the calculus of inductive constructions [32] (a kind of typed lambda calculus), this example is precisely in our target area of using XML for sophisticated language representations. Incidentally, it is a recursive DTD, not covered by Levene and Wood's theoretical result [11].
- *gene* contains genome data from the National Center for Biotechnology Information,[7] which offers a wide variety of biomedical and bibliographic data in a highly regimented XML format. We typed 'blood' into the search field and arbitrarily chose one of the roughly three thousand results to save as an XML file.
- *movies* contains data on movies and actors from IMDB, converted to XML by the World-Wide Web Wrapper Factory [33]. Unlike the NCBI examples, it is relatively straightforward markup that lists about a hundred actors and ten films.
- *pubmed* is another sample from NCBI, this time representing bibliographic data. We typed 'database' into the search field, and arbitrarily chose one of the roughly eighty thousand records.
- *hamlet* is, of course, from the previously-mentioned Shakespeare corpus.
- *sigmod* contains bibliographic data from ACM SIGMOD.[8]
- *uwm* contains course catalog data from University of Wisconsin, Milwaukee, obtained from the University of Washington repository.[9]
- *xmark* is a unit-size (about 100M) synthetic auction database produced by the `xmlgen` tool [34]. As one of the most widely-used XML benchmarks, it is helpful for comparing with published results about other systems.

[6]`http://coq.inria.fr/V8.1/stdlib/Coq.Lists.ListSet.html#set_mem_ind2`

[7]`http://www.ncbi.nlm.nih.gov/`

[8]`http://www.acm.org/sigs/sigmod/record/xml/`

[9]`http://www.cs.washington.edu/research/xmldatasets/www/repository.html`

Document Type Definitions were available for all of these data sets, and the documents validated with at most some minor tweaks. We converted the DTD format to Relax NG using `trang`.[10] We attempted to use some native Relax NG formats, but none of them worked out. For example, OpenLaszlo is a language for rich client-side applets, whose syntax is specified with Relax NG. Unfortunately, none of the validators we used (including, most importantly, Bali) could validate simple examples against the published schema. We hope to resolve these issues and test with native Relax NG formats in future work.

To clarify the results shown in figure 5, the bars show the compressed size of each data set relative to the size produced by `gzip` (at 1.0). The original uncompressed size of each file appears below its name in the graph. Lower bars are better, and our technique is the last bar in each group. We found that `rngzip` performed exceptionally well – better than any other compressor we tried – for the *coq, gene,* and *pubmed* data sets. It was one of the best on *movies* (with some tweaking of options, it can beat `xaust`, but as shown it loses by a few bytes). On *sigmod* and *xmark,* we are second only to `xml-wrt`. On *hamlet* and *uwm* our tool is slightly behind, but significantly better than previously reported.

Some missing bars deserve explanations. The (beta version) `dtdppm` crashed with a segmentation fault on *sigmod* and on any *xmark* data set. The `xaust` implementation is currently limited by not supporting ENTITY definitions in the DTD. For simple cases, entities can be removed using search-and-replace; we modified the *coq* and *sigmod* data sets so that it could participate. However, we decided that the *gene* and *pubmed* schemata were too complex to fix for the benefit of `xaust`. Otherwise, the performance profile of `xaust` is fairly similar to ours, in this sense: it does reasonably well on more data-oriented schemata, and is weaker on lightly-tagged text.

One factor in the effectiveness of our technique is the ratio of tags to text in the XML source. This number is provided across the lowest line of the bar graph (r). The *gene* schema is almost ludicrously 'taggy.' Here is how a time-stamp is represented:

```
<Date>
  <Date_std>
    <Date-std>
      <Date-std_year>2006</Date-std_year>
      <Date-std_month>4</Date-std_month>
      <Date-std_day>7</Date-std_day>
      <Date-std_hour>18</Date-std_hour>
      <Date-std_minute>55</Date-std_minute>
      <Date-std_second>0</Date-std_second>
    </Date-std>
  </Date_std>
</Date>
```

We do particularly well on this because the long tag names are stored in the shared schema instead of the compressed output, and the tag structure is very regimented. On data sets such as *hamlet* that consist more of lightly tagged text, our performance is predictably worse.

This ratio, however, is not the entire story. The *pubmed* set has a tag-to-text ratio in the same neighborhood as *hamlet*, but `rngzip` performs far better on *pubmed.* This difference is explained by the structure of the schema. The tags in Shakespeare are somewhat few and repetitive (`speech speaker line line line...`) which is good for text-based compressors, but there are technically many choices on every line (another line, end of speech, end of scene, end of act, ...) which is a disadvantage for schema-based systems. Nevertheless, the schema-based techniques measured here (`rngzip`, `xaust`, and `dtdppm`) perform better than the differential DTD compression of Sundaresan and Moussa [8], which could not even compress Hamlet because of this property.

The performance of `xml-wrt` on the large *xmark* file is astonishing. We verified that it decompressed properly, to ensure there was not some mistake. Indeed, it reproduced precisely the same file, including all the white space it could have ignored. We suspect that what happened here is that, although *xmark* is a large file, it is produced by `xmlgen`, a relatively small C program (about 430K source). Embedded in that program is a dictionary of all the words it will ever use. As a word-based compression technique, `xml-wrt` then builds a dictionary of words seen in the document, so it is essentially learning `xmlgen`'s dictionary. Its compressed representation of 1.8M is impressive – nobody else comes close – but we know there is a strict upper bound on the Kolmogorov complexity of *xmark:* the size of its source.

In the previous section, we detailed all the options available for encoders and compressors. The settings we selected for the numbers in the graph were `-E byte -T bz2 -Z ppm`; it was the best average performer over all eight benchmarks, even though it strictly won just on *xmark* and *coq.* The byte coder and PPM data compressor were fairly consistently the best, occasionally bested by fixed and bz2. Performance was less sensitive to the setting for the tree compressor. The adaptive Huffman coder was a complex implementation, and never worth it in the end.

We have not mentioned run-time performance yet, and with good reason! Such measurements seem unfair, since our tool is a rather large Java program, and all the others are written in C. We tested on a 1.8 GHz Intel Xeon with 768M RAM running Linux 2.6.18 and the Sun Java VM 1.5.0. We consider just the *gene* benchmark as representative. Elapsed times for most compressors were negligible: 0.02s for gzip, 0.10s for bzip2, etc. Xaust took 0.43s and xml-wrt 2.15s. Remember, however, that xml-wrt intentionally sacrifices compression speed for decompression. The elapsed compression time for `rngzip` on *gene* using the settings from the previous paragraph was 0.54s.

This seems reasonable, but that is strictly the compression time: it excludes the time it takes to build the

[10] `http://www.thaiopensource.com/relaxng/trang.html`

tree automaton from the schema. (The compressor and decompressor must both do this.) Building the automaton for gene takes another 6.28s. However, that could be done offline and either cached for fast loading, or we could generate specialized code (in any given language) to compress and decompress that particular schema more efficiently. In fact, `xaust` does exactly that. It processes the DTD offline and generates C code that then gets linked in with the tool itself. Similarly, Bali was originally designed to generate specialized validators in Java, C++, or C#.

V. Discussion and future directions

It is clear from the previous section that `rngzip` is most effective on highly tagged, nested data. The *coq* benchmark in particular is encouraging, because we aim to use this technology for producing compact representations of sophisticated intermediate languages [17] including ones based, like Coq, on the Calculus of Constructions [35].

In a previous incarnation of this work, our algorithm performed poorly on lightly tagged text, such as Hamlet and the course catalogs. This has improved considerably by incorporating some of the techniques of `xmlppm`, although we do not consistently match its performance on those data sets. It appears from these latest results that a best-of-breed XML compressor might also feature a word-replacing transform, like `xml-wrt`.

We previously discussed many other competing efforts on compressing XML data, but another piece of work deserves mention, from the domain of intermediate language representation. Amme et al. [36] encode programs using two innovative techniques: referential integrity and type separation. The effect is similar to what we do with XML, in that type-incorrect programs cannot be encoded at all. The compressed data are well-formed *by virtue of* the encoding.

In section III, we mentioned a certain property of Relax NG that conflicts with the goal of streaming. Within an element specification, attributes and child elements can be intermixed; the attributes need not appear first. Here is an example where the element `<top>` has some 'big' content; imagine megabytes of data and enormous subtrees.

```
top = element top { big, opt }
opt = attribute opt { text }
    | element opt { ... }
```

Following that, there is a *choice* between an `<opt>` element and an `opt` attribute (which applies to the parent element, `<top>`). What this means for decompressing is that we cannot output the `<top>` element until we know the result of the `opt` choice point, but that comes *after* the megabytes of data that must now be buffered. There is probably a work-around in the form of an automaton transformation that brings all the attribute transitions to the front. Martens et al. [37] provide formal evidence supporting this possibility: "[in the] tree grammars of Murata et al. [14]. . . every element in a document can be typed when its opening tag is met."

In addition to `text` and fixed strings used in figure 1, Relax NG can validate content using external libraries, such as the data type component of XML Schema. Our tool recognizes uses of such data types, but for now still treats them as though they were plain text. We intend in the future to use more compact encodings for content such as dates, *n*-bit integers, and base-64 binary data.

Another interesting extension – required by `xaust`, and related to Bali's validatelet capability – is to generate the code for a compressor and decompressor specialized to a particular schema. This could be a real advantage for applications that save their documents and data in XML-based formats. We hope that this line of work – on compact representations of XML – ultimately spells the end of the era of custom binary formats for storing or transmitting data.

Acknowledgments

We thank the referees and attendees of the IEEE Data Compression Conference for many enlightening comments on both the work and the presentation. Special thanks go to Kohsuke Kawaguchi, for making available key software libraries that significantly reduced our implementation task. These include Bali, the Sun Multi-Schema Validator, ISO-RELAX (with ASAMI Tomoharu, Daisuke Okajima, and MURATA Makoto), and the Relax NG Datatype library (with James Clark). We benefited also from the Colloquial Arithmetic Coder and PPM library by Bob Carpenter, the Apache Commons BZip2 library by Keiron Liddle, and the LZMA SDK by Igor Pavlov. Of course, any shortcomings in our system are our responsibility, and should not reflect on these authors' reputations.

References

[1] C. League and K. Eng, "Type-based compression of XML data," in *Proc. Data Compression Conference*. IEEE, March 2007, pp. 273–282.

[2] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan, "XML 1.1 (second edition)," Aug. 2006, W3C recommendation.

[3] H. Liefke and D. Suciu, "XMill: an efficient compressor for XML data," in *Proc. ACM SIGMOD Conference*. ACM, 2000, pp. 153–164.

[4] J. Cheney, "Compressing XML with multiplexed hierarchical PPM models," in *Proc. Data Compression Conference*, 2001.

[5] J. Adiego, G. Navarro, and P. de la Fuente, "Using structured contexts to compress semistructured text collections," *Information Processing and Management*, 2007.

[6] M. Girardot and N. Sundaresan, "Millau: an encoding format for efficient representation and exchange of XML over the Web," *Computer Networks*, vol. 33, pp. 747–765, 2000.

[7] P. Skibiński, S. Grabowski, and J. Swacha, "Effective asymmetric XML compression," February 2007, draft journal submission provided by authors.

[8] N. Sundaresan and R. Moussa, "Algorithms and programming models for efficient representation of XML for Internet applications," *Computer Networks*, vol. 39, pp. 681–697, 2002.

[9] P. M. Tolani and J. R. Haritsa, "XGRIND: A query-friendly XML compressor," in *Proc. Int'l. Conf. on Data Engineering (ICDE'02)*. IEEE, 2002, pp. 225–234.

[10] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, "Compressing and searching XML data via two zips," in *Proc. Int'l Conf. on the World Wide Web*. ACM, May 2006, pp. 751–760.

[11] M. Levene and P. Wood, "XML structure compression," in *Proc. 2nd Int'l Workshop on Web Dynamics*, 2002.

[12] J. Clark and M. Murata, "Relax NG specification," Dec. 2001, OASIS Committee.

[13] D. C. Fallside and P. Walmsley, "XML schema part 0: Primer (second edition)," Oct. 2004, W3C recommendation.

[14] M. Murata, D. Lee, M. Mani, and K. Kawaguchi, "Taxonomy of XML schema languages using formal language theory," *ACM Trans. Internet Technology*, vol. 5, no. 4, pp. 660–704, Nov. 2005.

[15] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra, "The Zephyr abstract syntax description language," in *USENIX Conf. on Domain-Specific Languages*, 1997, pp. 213–227.

[16] C. Brabrand, A. Møller, and M. I. Schwartzbach, "Dual syntax for XML languages," *Information Systems*, 2007.

[17] Z. Shao, C. League, and S. Monnier, "Implementing typed intermediate languages," in *Proc. Int'l Conf. Functional Programming*. ACM, Sept. 1998, pp. 313–323.

[18] P. Deutsch, "DEFLATE compressed data format specification, version 1.3," May 1996, RFC 1951, Aladdin Enterprises.

[19] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Transactions on Communications*, vol. 32, no. 4, pp. 396–402, April 1984.

[20] V. Toman, "Syntactical compression of XML data," in *Proc. Int'l Conf. on Advanced Information Systems Engineering*, 2004.

[21] P. Skibiński, J. Swacha, and S. Grabowski, "A highly efficient XML compression scheme for the web," September 2007, draft conference submission provided by authors.

[22] K. Chernik, J. Lánský, and L. Galamboš, "Syllable-based compression for XML documents," in *Proc. Int'l Workshop on Databases, Texts, Specifications, and Objects*, V. Snášel, K. Richta, and J. Pokorný, Eds., 2006, pp. 21–31.

[23] J. Lánský and M. Žemlička, "Text compression: Syllables," in *Proc. Int'l Workshop on Databases, Texts, Specifications, and Objects*, K. Richta, V. Snášel, and J. Pokorný, Eds., vol. 129, 2005, pp. 32–45.

[24] J. Cheney, "An empirical evaluation of simple DTD-conscious compression techniques," in *Eighth Int'l Workshop on the Web and Databases*, June 2005.

[25] H. Subramanian and P. Shankar, "Compressing XML documents using recursive finite state automata," in *Implementation and Application of Automata*, ser. LNCS, vol. 3845. Springer, 2006, pp. 282–293.

[26] S. Harrusi, A. Averbuch, and A. Yehudai, "XML syntax conscious compression," in *Proc. Data Compression Conference*. IEEE, 2006, pp. 402–411.

[27] ——, "Compact XML grammar-based compression," April 2007, draft manuscript provided by authors.

[28] S. Bressan, M.-L. Lee, Y. G. Li, Z. Lacroix, and U. Nambiar, "The XOO7 benchmark," in *Proc. VLDB Workshop on Efficiency and Effectiveness of XML Tools and Techniques*, ser. LNCS, vol. 2590. Springer, 2003, pp. 146–147.

[29] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, "Tree automata techniques and applications," 2002, http://www.grappa.univ-lille3.fr/tata/.

[30] D. E. Knuth, "Dynamic Huffman coding," *Journal of Algorithms*, vol. 6, no. 2, pp. 163–180, June 1985.

[31] A. Asperti, L. Padovani, C. S. Coen, F. Guidi, and I. Schena, "Mathematical knowledge management in HELM," *Annals of Mathematics and Artificial Intelligence*, vol. 38, no. 1-3, pp. 27–46, May 2003.

[32] T. Coquand and G. Huet, "The calculus of constructions," *Information and Computation*, vol. 76, pp. 95–120, 1988.

[33] A. Sahuguet and F. Azavant, "Building light-weight wrappers for legacy web data-sources using W4F," in *Proc. Int'l Conf. on Very Large Data Bases*. Morgan Kaufmann, 1999, pp. 738–741.

[34] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "Assessing XML data management with XMark," in *Proc. VLDB Workshop on Efficiency and Effectiveness of XML Tools and Techniques*, ser. LNCS, vol. 2590. Springer, 2003, pp. 144–145.

[35] C. League and S. Monnier, "Typed compilation against non-manifest base classes," in *Proc. Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, ser. Lecture Notes in Computer Science, vol. 3956. Springer, April 2006.

[36] W. Amme, N. Dalton, J. von Ronne, and M. Franz, "SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form," in *Proc. Conf. on Programming Language Design and Implementation*. ACM, 2001.

[37] W. Martens, F. Neven, and T. Schwentick, "Which XML schemas admit 1-pass preorder typing?" in *Proc. Int'l Conf. on Database Theory*, ser. LNCS, vol. 3363. Springer, 2005, pp. 68–82.

Christopher League was born in Baltimore, Maryland, in the early 1970s. He holds three degrees in computer science: a BS (1995) from Johns Hopkins University in Baltimore, an MS (1997) from University of Maryland, College Park, and a PhD (2002) from Yale University.

He is currently an Assistant Professor at Long Island University's Brooklyn Campus in New York City. His usual research interests lie in the area of programming languages and compilers, particularly on the role language technologies play in software reliability and computer security. Representative publications include "MetaOCaml server pages: web publishing as staged computation," *Science of Computer Programming* 62(1), Sept. 2006; and "Type-preserving compilation of Featherweight Java," *Transactions on Programming Languages and Systems* 24(2), March 2002. This is his first work on compression.

Prof. League received the department award for teaching excellence at the University of Maryland in 1997. He is a member of ACM SIGPLAN and the IEEE Computer Society.

Kenjone Eng is from New York City. He received his BS (2004) and MS (2007) degrees in computer science from Long Island University's Brooklyn Campus. Benchmarking `rngzip` was his MS thesis project.

While a student at LIU, he served as president of the CS Club, a student chapter of the ACM. Mr. Eng now works in information technology at LIU's University Center in Brookville, NY.