SML

A simpler and shorter representation of XML data inspired by Tcl

Jean-François Larvoire,
9 Chemin des Gandins, 38660 Saint Hilaire du Touvet, France
jf.larvoire@free.fr

2013-09-24

Abstract

XML is now the most popular standard for representing structured data as text.

Yet, despite all its successes, XML has failed one of its original design goals: Being easy to read and edit by human beings.

To address this problem, many others have proposed alternatives to XML. Some are indeed better, being both simple and more powerful. Yet I think they're missing the point. The benefits of having united the industry under a common data exchange standard far out weight the weaknesses of XML itself. It would be pointless to propose any alternative incompatible with XML now.

This paper proposes instead a Simplified representation of XML data (SML for short), inspired by the Tcl syntax, that is strictly equivalent to XML. But SML data files are smaller, and much easier to work with by mere humans.

A Tcl script called sml is available for converting files back and forth between the XML and SML formats.

The original idea was to use this script for converting XML files into this SML format; read them or edit them using a plain text editor; and convert them back to XML.

But other unexpected benefits came out of this Simplified XML representation:

- SML files are noticeably smaller than XML files. Using this format directly for storage or data transfer protocols would save space or network bandwidth.

- Using SML for data serialization is easier to work with than with XML, while future-proofing the compatibility with tools that know only of XML. Actually the SML format is so readable that I started using it as the native output format of all the management tools I wrote that output structured data… Some of which later got reused as XML input to other tools having no knowledge about SML.

- SML is a nice format for reviewing small file system trees contents, for example the Linux /proc/fs trees.

Introduction

We increasingly have to deal with XML files.

I started thinking about alternative views into XML files because of a personal itch: I needed to repeatedly tweak a complex XML configuration file for a Linux Heartbeat cluster in the lab. No DTDs available. No specialized XML editors installed on that machine. Editing the file using a plain text editor was painful every time.

Why had it to be so? XML is a text format that was supposed to be designed for easy manual edition by humans. And XML proponents actually list this feature as an advantage of XML. Yet XML tags are so verbose that it is a pain to manually review and edit anything but trivial XML files. The numerous XML editors available are a relief, but do not resolve the fundamental problem of XML verbosity when it comes to simply reading the file. (Actually I think their very existence is proof that XML has a problem!)

In the absence of a solution, I avoided using XML for my own projects as much as I could, and kept looking at alternatives, in the hope that one of them would eventually replace XML as the new data exchange standard.

Alternatives to XML

Many other people have complained about XML unfriendly syntax too, and many have proposed alternatives. Simply search "XML alternatives" of the Web and you'll find plenty!
(One of which was actually called SML too! No resemblance to this one).

A few important ones are:

- ASN.1 XER (XML Encoding Rules) [2]
  Pro: Powerful (more that XML). XER documents compatible with XML document model.
  Con: An adaptation of ASN.1 compatible with ASN.1 text format, but not with XML text format. i.e. conversion back to XML cannot yield identical files.
- JSON JavaScript Object Notation [3]
  Pro: Powerful (more that XML) and simple. Easy to use, with I/O libraries available for most languages. The most popular of the alternatives now, by far.
  Con: Completely incompatible with XML.
- Google Protocol Buffers [8]
  Pro: Simple syntax. Compiler for generating compact and fast binary encodings for wire transfers.
  Con: Even Google seems to prefer JSON for end-user APIs.

Some others have also attempted to "fix" XML by keeping only a subset of XML, for example by abandoning attributes (Ex: Simple XML [12]). Although this does make the tree structure simpler, this definitely does not make the document more readable. Even the W3C has made a proposal to simplify their own baby, also called Simple XML [11], although it does not go as far as the previous ones.

And in all cases backwards compatibility is lost.

Finally, there were even a few proposals made on the tcl.tk wiki:

- Xmlgen [13]
  Only designed to make it simple to generate XML, not as an alternative.
- TDL [14]
  Very similar to SML in many respects.
  Pro: Closer to the Tcl syntax than what I propose, and easier to parse as pure Tcl.
  Con: Not binary compatible; Less human friendly syntax for text, cdata, comments, etc.

Alternative views of XML

At the same time, I was writing Tcl scripts for managing Lustre file systems on that cluster. The instances of my scripts on every node were exchanging increasingly big Tcl structures (As strings, embedded in network packets), for synchronizing their action. And I kept finding this both convenient, and easy to program and debug. (ie. Review the structures exchanged when something goes wrong!)

And then I began to think that the two problems were linked: XML is nothing more than a textual presentation of a structured tree of data. A Tcl program or a Tcl data structure is also a textual presentation of a structured tree of data. And the essence of XML is not its <tagged><blocks>, but rather its logical structure with a tree of elements, attributes, and content blocks with other embedded elements inside. In other words its DOM (Document Object Model).

All programs written in C, Java, Tcl, PHP, etc, share a common simple syntax for representing program trees {based on {nested blocks} surrounded by parentheses}, which is much easier to read by humans than the <tagged><blocks> used by XML</blocks></tagged>. The Tcl language has the simplest syntax of them all, with a grammar with just a dozen rules. This makes it particularly easy to read and parse. And its one-instruction-per-line standard is a natural match to all modern XML files with one element per line.

Instead of reinventing a new data structure presentation language, it should be possible to convert XML into an equivalent Tcl-like format, while preserving all the elements, attributes, and data structures.

This defined a new problem: Find a text format inspired by Tcl, which is simpler than XML, yet is strictly equivalent to it. Equivalent in the mathematical sense that any XML file can be converted to that simpler format, then back into XML with no change whatsoever.

Non-goals: We do not try to generate a valid Tcl list of lists.

The SML Solution

Keep the XML data tree model with elements made of a tag, optional attributes, and an optional data block, but use a simpler text representation based on the syntax of C-style languages.

The basic idea is that XML and SML elements correspond to each other like this:

- XML elements: <tag attribute="value" ...>contents</tag>
- SML elements: tag attribute="value" ... {contents}

But the devil lies in the details, and it took a while to find a set of rules that would cover all XML syntax cases, allow fully reversible conversions, optimize the readability of real-world files, and remain reasonably simple. After experimenting with a number of alternatives, I arrived at the set of rules defined further down, which give good results on real-world documents. Example extracted from a Google Earth file:

XML (from a Google Earth .kml file)

```
<?xml version="1.0" encoding="UTF-8"?>
<kml>
  <Folder>
    <name>Take off zones in the Alps</name>
    <open>1</open>
    <Folder>
      <name>Drome</name>
      <visibility>0</visibility>
      <Placemark>
        <description>Take off</description>
        <name>Mont Rachas</name>
        <LookAt>
          <longitude>5.0116666667</longitude>
          <latitude>44.8355</latitude>
          <range>4000</range>
          <tilt>45</tilt>
          <heading>0</heading>
        </LookAt>
      </Placemark>
    </Folder>
  </Folder>
</kml>
```

SML (generated by the sml script)

```
?xml version="1.0" encoding="UTF-8"
kml {
  Folder {
    name "Take off zones in the Alps"
    open 1
    Folder {
      name Drome
      visibility 0
      Placemark {
        description "Take off"
        name "Mont Rachas"
        LookAt {
          longitude 5.0116666667
          latitude 44.8355
          range 4000
          tilt 45
          heading 0
        }
      }
    }
  }
}
```

The difference in readability should be immediately obvious!

SML Syntax rules

Elements

- Elements normally end at the end of the line.
- They continue on the next line if there's a trailing '\'.
- They also continue if there's an unmatched "quotes" or {curly braces} block.
- Multiple elements on the same line must be separated by a ';'.

Attributes

- The syntax for attributes is the same as for XML. Including the rules for using quotes and escape chars. (And so is different from Tcl's string quoting rules.)
- There must be at least one space between the last attribute and the beginning of the content data.

Content data

- The content data are normally inside a {curly braces} block.
- The content text is between "quotes". Escape '\' and '"' with a '\'.
- If there are no further child elements embedded in contents (i.e. it's only text), the braces can be omitted.
- Furthermore, if the text does not contain blanks, '"', '=', ';', '#', '{', '}', '<', '>', nor a trailing '\', the quotes around the text can be omitted too.
  (ie. If the text cannot be confused with an attribute or a comment or any kind of SML markup.)

Other types of markup

All use the same rules as the elements for juxtaposition and continuation.

- This is a **?Processing instruction** . (The final '?' in XML is removed in SML.)
- This is a **!Declaration** . (Ex: a !doctype definition)
- This is a **#-- Comment block, ending with two dashes --** .
- Simplified case for a **# One-line comment** .
- This is a **<[[Cdata section]]>** .
  An optional new line, immediately following the opening <[[, is discarded if present.

Heuristics for XML<->SML conversion

- Spaces/tabs/new lines are preserved.

- The sml program adds one space after the end of the element definition (ie. after the last attribute and optional trailing spaces inside the element head), before the beginning of the data block.
  This considerably improves the readability of the sml output.
  Then it removes it when converting SML back to XML.
  An SML file is invalid without that space anyway.
- Empty data blocks (i.e. Blocks containing just spaces) encoding: Use {} for multiline blocks, and "" for single-line ones.
- Unquoted attribute values are accepted, in an attempt to be compatible with HTML-style attributes, which do occur in poorly-written XML files.

Syntax rules discussion

Except fo XHTML, real world XML files usually contain a hierarchy of outer elements, with 0 or more inner elements, but no text. Then the terminal elements (the inner-most elements) contain just text.

- *SML elements normally end at the end of the line*.
  A natural match for most modern XML files, which usually have one XML terminal element per line.
- *They continue on the next line if there's a trailing '\'.*
  Same rule as for Tcl, and many other programming languages.
- *They also continue if there's an unmatched "quotes" or {curly braces} block*.
  This is a major advantage of the Tcl syntax, allowing to minimize the syntactic glue characters.
- *Multiple elements on the same line must be separated by a ';'.*
  Again, the same as Tcl.
- *The syntax for attributes is the same as for XML:* name="value"
  XML attribute name ::= (Letter | '_' | ':') (Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender)*
  XML attribute value ::= ("" ([^<&"] | Reference)* "") | ("'" ([^<&'] | Reference)* "'")
  Use references like & , < , > , ' , " to quote the special characters in values.
  I considered using Tcl's quoting rules instead. But this made the conversion program more complex, and did not make the SML more readable. (Actually it made it less readable, making it more difficult to read long lists of attributes.)
  Most real-world attribute values will look exactly the same as the equivalent Tcl string anyway.
  => I'm open to changing this rule for compatibility with Tcl quoting.
  Also TDL [14] proposes an interesting alternative: Write attributes are Tcl functions options with a dash: -name value
  Pro: Easier to parse as a Tcl list.
  Con: Less intuitive to people who don't know Tcl.
  Con: Makes it more difficult to deal with HTML-like attributes that have no value.
- *The content data are normally inside a {curly braces} block. Braces in the content text must be escaped by a '\'.*
  Same as Tcl {blocks}. Works well for XML outer elements containing inner elements.
- *If there are no further child elements embedded in contents (i.e. only text), a "quotes" block should be used instead. In that case, braces in the text need not be escaped, but the '\' and '"' should be instead.*
  Works well with terminal elements containing just text.
- *The quotes around text can be omitted if the text does not contain blanks, '"', '=', ';', '#', '{', '}', '<', '>', nor a trailing '\', and if there are no other elements at the same tree depth. (ie. It cannot be confused with an attribute or a comment or any kind of SML markup.)*
  Maximizes readability by removing all extra characters around simple values.

Possible alternative: In the cases where text and elements are mixed at the same tree depth (rare, except in XHTML), use a pseudo element tag like !text or just @ (But not #text which would look like a comment) to flag it. This would allow extending the SML syntax to support element names with spaces. See the "show script" chapter below for a useful application of that.

- *This is a* ?Processing instruction .
  This is a !Declaration . (Ex: A !doctype definition)
  Both are treated like XML empty elements, with a name beginning with an '?' or a '!'.
  All contents are preserved, except for the final ?> and > respectively.
  Add a '\' at the end of lines if the element continues on the following lines.

- *Simplified case for a* # One-line comment .
  Same as for Tcl, and many other scripting languages.

- *This is a* #-- Comment block -- .
  I considered using existing syntaxes, like <# Multiline comment #> in PowerShell. But this was barely more concise, and this created problems to deal with the -- sequence in SML (not valid in an XML comment), or the #> sequence in XML (not valid in an SML comment in that case)
  In fine, the simplest was to stick to the -- delimiters like in XML.

- *This is a* <[[CDATA section]]>
  Like for comment blocks, sticking to the XML termination sequence proved to be the easiest option. Any other type of delimiter would have required complex escaping rules, for the case that delimiter appears in the CDATA itself. The possibility of having adjacent CDATA sections would have made these rules even more complex.
  By symmetry, I used `<[[` for the opening sequence.
  Note that the CDATA `]]>` end markers cannot be confused with the `]]>` end markers at the end of some complex !declarations, because those ones become `]]` after the final '>' is removed in SML.

  An optional new line, immediately following the opening <[[, is discarded.
  This makes it easy to view multiple lines of CDATA. The first line will begin on the first column, like all the others.
  Gotcha: That additional new line must be inserted if the CDATA begins with an initial new line. Else the initial new line would be lost during the conversion back to XML.

  Possible alternative: I experimented with simpler alternatives in other programs.
  One is the indented block, used in the "show" program described further down:

```
Preceding content{
  This is a sample CDATA with an XML <tag>
}Following content
```

  Here, the rule is that all CDATA block contents are indented by two more spaces than the previous line.
  The first '}' at the same indentation as the opening '{' sign marks the end the CDATA.
  The CDATA begins after the new line following the opening '{' (So this new line is not optional here), and ends before the final new line before the closing '}'.
  Pro: More lightweight syntax, more in the spirit of Tcl.
  Pro: Looks better in deep trees, as multi-line CDATA blocks are indented like the rest.
  Con: Adds numerous spaces, and makes the CDATA block weight more in bytes.
  Con: Made the sml conversion program more complex and slower.

  Variation on the same theme: Particular case of a CDATA section that makes up the whole content of an element:
  Instead of encoding this content block with double parenthesis `{{\n  CDATA\n}}`, it'd be written `={\n  CDATA\n}`

The sml script

A working proof of concept program, called sml.tcl, is available at URL:
http://jf.larvoire.free.fr/progs/sml.tcl

It works in any system with a Tcl interpreter. (Standard in Linux. A free version for Windows is available at http://www.activestate.com/activetcl)

I've tested it on a number of sample XML files from various sources, totaling about 1 MB.

It is able to convert them all to SML, then back into XML, with the final XML files binary equal to the originals.

A simple glance at the contents of the SML files will show, as in the example above, that the "useful" information is much easier to find. The eye is not distracted anymore by the noise of useless end tags and brackets.

Current Status

The proof of concept is written in Tcl, and works fine on my sample files. I've been using it regularly for several years.

The file has about 3000 lines of code, half of which are an independent debugging library.

The only issue is performance: It converts about 10 KB/s of data on a 2 GHz machine. Rewriting it in C and optimizing the lowest I/O routines should be able to increase performance by several orders of magnitude.

Actually the best bet would be to start with an XML parsing library, and modify it to parse or generate either XML or SML. This should be easy since the only difference is the tree representation, not the DOM.

I've not attempted to use Tcl's own XML parsing libraries, like TclDOM or tDOM, mostly for lack of time, and because the performance on small to medium XML files is adequate for me. Also I'm a bit concerned about the way spaces are handled by these libraries. They must be preserved and provided to the caller for reversibility. But maybe they are already?

Known limitations:

- The converted files use the local operating system line endings (a combination of \r and \n). So if the initial XML file was encoded with line endings for another operating system, converting it to SML then back will not be binary equal to the initial file. But it will still be logically equal, as the XML spec states that all line endings are equivalent to \n.
- As of 2013-09-22, complex !doctype declarations with locally defined !elements are not supported.

Sml files size

An interesting side benefit of the conversion is that the total size of the converted files is 12% smaller than the original XML files. Among big files, that reduction goes from 4% for a file with lots of large CDATA elements, to 17% for a file with deeply nested elements.

Even after zipping the two full sets of samples, the SML files archive is 2% smaller than the XML files archive. Not much I admit, but this may help Microsoft alleviate the new Office documents bloat. ☺

As for XML compression, many dedicated compressors are available (Ex [4], [5]). Obviously they give better results than SML. But just as obviously the compressed files are unreadable by humans!

Reductions are much better on xml documents using name spaces. For example on the sample SOAP envelope from the SOAP 1.2 specification, the gain is 30%. Transporting SOAP messages in their SML form instead of XML would yield huge network bandwidth gains! (In case somebody still uses SOAP! ☺)

The show script

This script allows displaying a file tree as experimental SML.

Available at URL: http://jf.larvoire.free.fr/progs/show.tcl

The principle is that each file or directory is an SML element. Directories contain inner elements that represent files and subdirectories. File contents are displayed as text if possible, else are dumped in hexadecimal.

It also has options for generating several alternative experimental SML formats, which have helped convince me which was the most readable solution.

The show script has two major modes of operation:

- A simplified mode, which is not fully SML-compatible, but produces the shortest output, easiest to read. (This is the default mode of operation)
- A strict mode, which produces a fully SML-compatible output, at the cost of a heavier output.

The simple file system representation

- The tag name is the file or directory name. (This is what is not XML compatible. XML requires tag names to begin by a letter and contain only alphanumeric characters plus '–' and '_'. Names containing spaces, or SML markup characters like a '#' in the first place, must be "quoted"… Which the sml script would consider as the beginning of text content, not an element.)
- Directory names end with a '/'.
- The file size, date/time, owner, etc, are SML attributes, not shown by default.
- Files contents are shown indented. (By default, only the first 10 lines are shown.)
- Text files are shown as they are, except for the indent.
  Binary files are shown as an hexadecimal dump.

Example under Linux:

```
[root@xena1 ~]# /k/Tools/show.tcl /proc/fs/cifs
/proc/fs/cifs/ {
  cifsFYI 0
  DebugData {
    Display Internal CIFS Data Structures for Debugging
    ---------------------------------------------------
    CIFS Version 1.68
    Active VFS Requests: 0
    Servers:
    1) Name: 10.16.131.25  Domain: LAB Uses: 1 OS: Windows Server 2008 R2
Enterprise 7601 Service Pack 1
        NOS: Windows Server 2008 R2 Enterprise 6.1      Capability: 0x1e3fc
        SMB session status: 1   TCP status: 1
        Local Users To Server: 1 SecMode: 0x3 Req On Wire: 0
        Shares:
  } ...
  LinuxExtensionsEnabled 1
  LookupCacheEnabled 1
  MultiuserMount 0
  OplockEnabled 1
```

```
  SecurityFlags 0x7
  Stats {
    Resources in use
    CIFS Session: 2
    Share (unique mount targets): 2
    SMB Request/Response Buffer: 2 Pool size: 6
    SMB Small Req/Resp Buffer: 2 Pool size: 30
    Operations (MIDs): 0

    9 session 1 share reconnects
    Total vfs operations: 1252680 maximum at one time: 2

  } ...
  traceSMB 0
}
[root@xena1 ~]#
```

The strict file system representation

- The tag name is the object type (Ex: "file" or "directory" or "symlink", etc.)
- The name is an SML attribute called name. (Ex: file name="SML specification.pdf")
- All other file properties, like date/time, modes, owner, access control lists, etc, are SML attributes.

This one is fully SML and XML compatible. And the textual output can be (in theory) used to recreate the complete file system.

The spath script

This script does not exist, but this chapter is a thought experiment that gives some insight on the power of the SML concept:

I had made another script called xpath.tcl, which makes it easy to use XPATH to view the contents of XML files, or extract data from them. This script does nothing fancy. All it does is to pretend the XML file represents a file system, and allow accessing its contents using Unix-style commands like cat or ls. XML elements are considered as directories, and attributes as files. The content data for a terminal element is considered as an unnamed file. Examples:

```
xpath sites.kml ls /kml/Folder/Folder
```

lists all inner elements as directories, and attributes as files.

```
xpath sites.kml cat /kml/Folder/Folder/name
```

Displays attribute values, or the text content for elements. Here it outputs “Drome”.

The idea here is to write an spath.tcl script that does the same for SML data instead of XML.

Supporting all features of XPATH would be difficult, as xpath.tcl uses Tcl’s TclDOM package to do the real work. But in the short term, it’s possible to get the same functionality using a one-line shell script:

```
sml | xpath %*          (Or $* for Unix)
```

1) This example shows the power of having a data format that is equivalent to XML.

2) Notice how this works nicely with the output of the show script described in the previous chapter: show.tcl

captures the contents of a real file system, where files are normally displayed with the `cat path` command. Then spath.bat allows extracting the contents of individual files from that sml file using `--cat path`. The `path` is the same. Gotcha: Unfortunately this does not work with file names that are not XML tag compliant, for example if they contain spaces, or begin with a digit, etc. A possible addition to XML 2.0 maybe? ☺

Exporting Tcl data

The SML data format would be a natural format for exporting any kind of Tcl data:
- SML looks a lot like Tcl itself. (Particularly when not using attributes.)
- In simple cases (without attributes), SML can usually be parsed by the Tcl interpreter directly as Tcl lists of lists.
- You get for free the compatibility with any outside tool that only supports XML.

Side note about Tcl name spaces

The convergence of XML element trees, file system trees, and variable name spaces, leads me to think that the natural syntax for name spaces should be /namespace/subspace/variable, not ::namespace::subspace::variable.

SML management libraries

An SML parsing library?

I never had to write one…

- In simple cases (often) SML just looks like tcl lists, which the tcl interpreter can digest directly.

- In complex cases (rarely), it was too easy to use the sml script to generate XML, then the TclDOM package to parse it. So I never took the time to write a full-fledged parsing library.

Of course, ~~for the sake of the art~~ for performance reasons, it would be nice to write one in C or C++, using APIs similar to XML parsing libs.

An SML generation library?

I never had needs that required a full-fledged library like what TclDOM or tDOM allows to do for XML.

For my simple needs, I usually generated the output recursively, with the terminal elements first, then a call to a very simple IndentString routine to indent a block, which becomes the content block of an outer element, etc. This works well enough for small output files. For bigger ones, with deep element indentation levels, this will not scale well, as the inner elements have to be indented over and over again.

Curiously the best SML generation routines I have are written in PowerShell. These routines use yet another file system paradigm variation, with enclosing elements thought of as directories, and terminal elements thought of as files. Also they can generate either XML or SML at the flip of a global (shame on me) switch:

Put-Dir dirname [attributes] [script_block]	Begins an element, and opens an indented block. If the script block is present, its output will be indented in the content block, then that block will be closed immediately afterwards.
Put-EndDir dirname	Closes a content block. (Not necessary when using a script block with Put-Dir.)

<table>
<tr><td></td><td>The dirname argument is only necessary for generating XML, and even then I could get rid of it with a little more work, by recording names in a stack.</td></tr>
<tr><td>Put-Value name value [attributes] [options]</td><td>Ouputs a terminal element. Options include:<table><tr><td>-nameWidth N</td><td>Force the width of the element name. Allows aligning several name/value pairs, making them easier to read.</td></tr></table></td></tr>
</table>

Rewriting these routines in Tcl would be easy. (In part due to the similarity of Tcl and PowerShell for passing script blocks as arguments.) In PowerShell, the attributes are passed in a hashtable. In Tcl they'd be in a dictionary.

Next Steps

- Let people experiment with the tools, and give feedback about the syntax, and the possible alternatives. Is there any error or inconsistency that remains, preventing full XML compatibility in some case?
- If interest grows, work with interested people to freeze a standard, and create a simple SML generation and parsing library.
- If interest grows even further, work with the TclDOM and tDOM developers to see if this could be added to their libraries as an alternative XML encoding format, for both input and output.
- Any project which stores data as XML files, even zipped like in MS Office, would save space and increase ease of use by using the SML format instead.
- The savings potential is even better in XML-based network protocols, such as SOAP. Adapting existing XML-based protocols to use SML instead would be very easy, and increase bandwidth considerably. Creating new ad-hoc SML-based protocols would be easy too, and packet analysis would be much easier!
- Any new project which does not know what data format to use, could get an easy-to-use format by adopting this SML format, while ensuring compatibility with XML-compatible-only tools, should the need arise.

References

[1] XML specification: http://www.w3.org/TR/xml/

[2] ASN.1 XER (XML Encoding Rules): http://asn1.elibel.tm.fr/xml/xer.htm
http://www.itu.int/ITU-T/studygroups/com17/languages/X.693-0112.pdf

[3] JSON JavaScript Object Notation: http://www.crockford.com/JSON/

[4] Open Mobile Alliance WBXML (Wireless Binary XML Encoding) Specification
http://www.openmobilealliance.org/tech/affiliates/wap/wap-192-wbxml-20010725-a.pdf

[5] Compressing XML with Multiplexed Hierarchical PPM Models
http://xmlppm.sourceforge.net/paper/paper.html

[6] A list of XML alternatives proposals: http://www.pault.com/xmlalternatives.html

[7] XML compression bibliography: http://xmlppm.sourceforge.net/paper/node9.html

[8] Google Protocol Buffers: https://code.google.com/p/protobuf/
http://google-opensource.blogspot.fr/2008/07/protocol-buffers-googles-data.html

[9] XML discussions on Tcl's wiki: http://wiki.tcl.tk/1740

[10] TDL proposal on Tcl's wiki: http://wiki.tcl.tk/25681

[11] Simple XML: http://www.w3.org/XML/simple-XML.html

[12] Simple XML (unrelated to the previous one): http://en.wikipedia.org/wiki/Simple_XML

[13] Xmlgen presentation on Tcl's wiki: http://wiki.tcl.tk/5976?redir=3210

[14] TDL proposal on Tcl's wiki: http://wiki.tcl.tk/25681