Test::XML::Generator
Generating XML for Unit Testing

Alberto Simões

Esc. Sup. de Est. Industriais e de Gestão
Instituto Politécnico do Porto
alberto.simoes@eu.ipp.pt

Abstract. To define a DTD or a Schema is not a trivial task. It can be compared to the task of preparing a data structure or, in some cases, to program that data structure adding some semantic. This makes this task error prone. It is common that a final Schema/DTD supports some special XML structures that should not be considered correct, or that, although these special structures are correct, they are not being correctly managed by the application parsing them.
We defend that the possibility to automatically generate XML documents based on a Schema/DTD can help preventing these situations. The generated documents can be used for unit testing and help tuning the Schema/DTD or fixing application problems. They can also assist on benchmarking issues as sometimes developers does not have access to full featured real world documents.
In this article we discuss a Perl module (`Test::XML::Generator`) that provides different mechanisms for automatically generate XML documents based on a DTD and a set of controlling parameters.

1 Introduction

While not trivial, the programming task is not properly difficult. Most problems are not related to the algorithm (unless you are programming in highly scientific areas) but to details that are not considered during the analysis and development task. This is often an issue of forgetting that real world situations can not be predictable. While common situations are easy to predict, edge situations are not.

When programming XML (eXtensible Mark-up Language) processors this is also true. As when dealing with relational databases where programmers need to take care of the database schema, when dealing with XML documents programmers need to take care of the XML structure, often specified on a DTD (Document Type Definition) or an XML Schema.

The process of understanding one of these documents is not a trivial task. Most DTD document can be easily understandable with some paper work and with a detailed analysis of the document grammar. While XML Schemas are not easy to read and understand directly from the XML syntax, there are a lot of

schema visualizers that makes this task easier. Although these documents can be analyzed easily, some details can and will always slip.

We defend that the ability to automatically generate XML documents covering all the edge cases of a XML DTD or Schema can help on debugging the applications that will deal with their instance documents.

Edge cases can include:

- lists with no elements
  should the DTD use a one-or-more list instead of a none-or-more list?;
- element with multiple optional child where none are present
  this is a relevant situation especially when dealing with DTD definitions as they lack flexibility for defining these cases;
- too deep element recursion or too big document size
  programmers often test applications with small and simple XML instances;
- missing attributes
  are the attributes required or optional?;
- conflicting attributes
  are there attributes that can not be used together? DTD also misses flexibility on this situation;

If we are able to generate one or more documents covering all or most of the structure defined in the XML DTD or Schemas, the programmer can test his application robustness during development time.

This article will detail a Perl Module, named `Test::XML::Generator`, that provides a simple programmers interface for generating controlled random XML documents based on a DTD[1].

First we will present briefly other tools with similar objectives. Then, a section documenting `Test::XML::Generator` architecture and algorithm, focusing on the tuning parameters to control the XML documents structure, size, depth and other. Follows a section on the tool analysis. Finally we will reflect and conclude about the tool usefulness.

2 Similar Tools

While there are some XML generator tools like Stylus Studio® 2010 XML, Visual Age for Java or Sun XML Instance Generator, these tools generate mostly simple (or dummy) XML documents, with no possibility to force their size or behavior. They are integrated in IDE tools and are used for small XML documents to be enlarged by the user, and not with testing purposes.

Other tools, like IBM XML Generator or ToXgene [1,2] use annotated templates (annotated DTD or annotated XML template, respectively) to generate the documents. While this approach is especially interesting as they generate semantically correct and controlled documents, they lack on generality and simplicity, as the user needs to understand the templating language, and understand

[1] While we are aiming at XML Schemas as well their structure is much more rich and needs further study and evaluation.

the DTD before writing the template. Therefore, the problems of forgetting corner cases will arise as well.

3 `Test::XML::Generator` Approach

`Test::XML::Generator` has three main approaches:

- **Random document generation** – the DTD structure is traversed and an XML document is generated randomly. Optional elements can be, or not, generated, lists will include a medium number of elements, and random content will be generated for each element. This approach tries to mimic the common XML documents that the applications will need to process.
- **Minimum document generation** – when generating the minimum document, the algorithm will choose the minimum content possible for each element. Optional elements will not be present, lists will include just one or no elements, depending on the list arity, and attributes will be present only if required.
- **Maximum document generation** – this approach is very similar to the first random document generation, but lists, repetitions and recursion will be forced to the limits. As recursion can, on some kind of documents, be infinite, `Test::XML::Generator` uses some variables to control maximum recursion and list size: one limit for list sizes (`LIST_SIZE`), one limit for recursion per element name (`TAG_DEPTH`), and one other to control full recursion size (`FULL_DEPTH`).

To help on the description of the algorithm, the following structure will be used: for each element or attribute type a list of items will describe the algorithm behavior for that element type in this order: random document (RAND), minimum document (MIN) and maximum document (MAX) generation.

At the present moment `Test::XML::Generator` reacts to the following child types:

Enumerated sequence of elements – An enumerated sequence of child elements has the same reaction for the three approaches as all children are required: each element is processed and the result is concatenated together:

- **RAND:** concatenation of processing all children;
- **MIN:** concatenation of processing all children;
- **MAX:** concatenation of processing all children;

Lists of elements – There are two kind of lists, the ones accepting zero elements and the others where one element is required. As already stated, the algorithm uses the parameter `LIST_SIZE` to control the maximum size of lists, and generates:

- **RAND:** a pseudo-random number of elements, ranging from 0 or 1 (depending on the list type) to `LIST_SIZE` The algorithm forces this value to lower with the recursion depth (again, trying to mimic most common XML documents), and helping to control the document size;
- **MIN:** one or no element depending of the list size;
- **MAX:** a list with `LIST_SIZE` elements, unless that would make any of the recursion depth control variables to overflow.

Optional elements – Optional elements are a special kind of lists where the maximum number of child elements is just one:

- **RAND:** one or no element will be generated. Probability to generate empty elements will raise with the recursion level;
- **MIN:** no element will be generated;
- **MAX:** one element is always generated unless that would make any of the recursion depth control variables to overflow.

Choice or Mixed Content Elements – Yet again, mixed content elements can be considered lists but, instead of homogeneous content, they support different kind of elements. These elements can be divided in just choice elements, where children are elements, and mixed content elements, where children are elements or textual content. In any algorithm if the element supports mixed content, then textual content will have the same probability to be generated as the sum of the probabilities of the other elements. This will force the element to include the same number of text and element children. All other elements will have same probability to be generated.

- **RAND:** a random number of children will be generated, ranging from 0 to $n \times$ `LIST_SIZE` where n stands for the number of the optional elements. Recursion control variables will be used to ensure the algorithm ends.
- **MIN:** no element will be generated;
- **MAX:** $n \times$ `LIST_SIZE` elements will be generated unless recursion limits are reached.

Text nodes – Text nodes will be filled with random text. In this case, the well known "*Lorem Ipsum*" text generator [3] is used.

- **RAND:** a paragraph of *Lorem Ipsum* is generated.
- **MIN:** one or two words are generated. `Test::XML::Generator` supports a special flag to tell the algorithm that empty text nodes are possible;
- **MAX:** a paragraph of *Lorem Ipsum* is generated.

Any Elements – These elements are similar to text nodes but can contain tags (defined or not in the DTD). At the moment the tool considers these elements are standard text nodes. Future versions will include some random generated tags, so applications can test their handling of unknown elements.

Empty Elements – These elements can not include any content in any case, therefore no further discussion is necessary about them.

Required attributes – These attributes will be always added for the elements they are defined, as no other option is possible.

Optional attributes – Option attributes are both `IMPLIED` and `DEFAULT` DTD attributes and will be:

- **RAND:** randomly generated (50% of probability for being generated);
- **MIN:** not be generated;
- **MAX:** always be generated.

Regarding the attribute generation some caution should be taken when generating the attribute value. DTD does not define much differences on the attributes values, defining a few choices. The more used kinds are the enumerated, `CDATA`, `ID` and `IDREF` (or `IDREFS`).

While the generation of content for the enumerated and `CDATA` types is simple, the generation of identifiers and their reference is not that easy. This is specially true because DTD does not specify semantic information about what elements can reference what elements. Therefore, `Test::XML::Generator` is only able to ensure every `IDREF` has an existing `ID` (semantically valid or not).

4 Testing `Test::XML::Generator`

The overall algorithm is relatively simple. The main problem resides on the structure and complexness of the supplied DTD.

The generation of minimum document are fast and reliable. That might be the reason most applications include that feature (check the conclusion for that discussion). The problem is with the generation of random documents. If the recursion depth is big, or the XML document complex, the process can take several minutes.

Consider the following simple yet recursive DTD:

```
<!ELEMENT x (bar, zbr)>
<!ELEMENT bar (zbr+)>
<!ELEMENT zbr (foo|bar)>
<!ELEMENT foo (#PCDATA)>
```

The generation of a random document with list sizes of 30 elements and maximum depth of 10 per tag, and 20 in total, can take 15 seconds to generate a 7MB file. This file includes 217 456 elements `zbr`, 54 304 elements `foo` and 20 615 elements `bar`, with a total of 292 376 elements.

While we could use other well known DTD like RSS, Atom XML or TMX, they are not recursive. Other more recursive standards have their structure defined in XML Schemas or in complex DTD documents, that would make tool experiments harder (different points of rupture). Nevertheless, as soon as the tool gets stable these tests are indispensable.

5 Conclusions

This tool is not yet ready and still has problems. At the moment it is mostly a proof of concept, showing the possibility of generating different kind of dummy documents with different testing goals.

Main problems on the document generation task is the generation of non-empty documents. Questions like "where to stop recursion" or "how many tags should be used in a repetition" should be addressed. While we can use some limits (as we did in `Test::XML::Generator`), the generation time is not predictable given the randomness of the algorithm.

Finally, it is our claim that XML generator should be promoted as a stand alone tool instead of a toy option from IDE applications.

Future Work

The presented tool is under alpha stage, and further tests are needed, mainly to find out what the best values to be used on the controlling variables, namely recursion ones.

It is expected that the DTD parser module handle automatically parametric entities, expanding them into standard DTD constructions. Unfortunately this was not yet performed.

When the generation process gets stable, the tool should be expanded to parse XML Schemas as well, as they include more semantic and type information that can be used during documents generation. As already stated, this type of document structure will raise a lot of problems given its flexibility, data types and complex construction operators.

References

1. Denilson Barbosa, Alberto Mendelzon, John Keenleyside, and Kelly Lyons. Toxgene: A template-based data generator for XML. In *In Proc. Fifth Intl. Workshop on the Web and Databases (WebDB)*, pages 6–7, 2002.
2. Denilson Barbosa, Alberto O. Mendelzon, John Keenleyside, and Kelly Lyons. Toxgene: An extensible template-based data generator for xml. In *In WebDB*, pages 49–54, 2002.
3. Wikipedia. Lorem ipsum — wikipedia, the free encyclopedia, 2010. [Online; accessed 12-March-2010].