The Xlint Project[*]

Juan Fernando Arguello, Yuhui Jin
{jarguell, yhjin}@db.stanford.edu
Stanford University
December 24, 2003

1 Motivation

Extensible Markup Language (XML) [1] is a simple, very flexible text format derived from SGML. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. XML documents can be huge, especially when they are converted from other formats such as HTML or relational tables. For example, Professor Gio Wiederhold at Stanford University maintains a relation movie database. An HTML version of the movie database was converted into XML, resulting in large files such as a 5.4 MB XML document containing all the movie information [2].

Due to the imperfect conversion process (especially in the case of manual conversion), many of the resulting XML documents are not well-formed, i.e., they contain errors such as wrong syntax for comments or attribute declarations, mismatch of tags, etc. According to the XML specification, violations of well-formedness constraints are fatal errors – once a fatal error is detected, the XML processor must not continue normal processing (i.e., it must not continue to pass character data and information about the document's logical structure to the application in the normal way). Due to this requirement, all existing XML parsers will stop right after the first well-formedness error is detected. To be able to use an XML document for any purpose, we need to first remove all the well-formedness errors. Using a conventional parser to detect and fix the errors causes us to repeatedly run the parser to detect and remove each error. This is very inefficient for a huge XML document with many errors (some of them may be systematic errors caused by improper global replacements). A desirable parser should never stop and report all the well-formedness errors during or after a complete parse of the XML document. The Xlint project aims at creating such an error-tolerant XML parser to facilitate the error removal of large XML documents.

2 XML Parsing Techniques

The most popular parsing techniques used by parsers today fall into the following three categories:

- Document Object Model (DOM)

[*] This work is supported US Air Force and the DARPA DAML project "OntoAgents" (01IN901C0).

- Simple API for XML (SAX)
- Streaming API for XML (StAX)

We briefly describe each parsing technique, its advantages and disadvantages.

2.1 DOM Parser

The DOM parser is a tree-based parser. A DOM parser will read in an XML document completely and construct a parse tree structure in memory. The root of the tree structure represents the XML document. It has at least one child node which is the root element tag of the document. All other element tags following the root element tag become child nodes in the tree structure. This tree structure must be loaded into memory first before the parsing can begin. The advantage of this parsing technique is that it allows dynamic access to the entire tree structure. This makes it possible for the application to navigate through only the interesting portions of the XML document. The disadvantage is that a DOM parser quickly becomes a costly choice when applied to large documents. Moreover, the DOM parser can not perform partial parsing – the entire XML document must be parsed at one time; for a document of considerable size, this may take up too much memory even if only a piece of the document is to be used.

2.2 SAX Parser

A SAX parser adopts an event-driven push model for processing XML documents. Instead of building a tree structure made up of a document's element tags, a SAX parser detects elements of the document through corresponding events, i.e., startElement, endElement, Characters, Attributes. These events are then pushed to event handlers where the application executes a certain task based on the event it receives. The three main types of event handlers are the following:

- DTDHandler – to access the DTD of an XML document
- ErrorHandler – to access any parsing errors, i.e., syntactical, structural
- ContentHandler – to access the contents within the XML document

Unlike DOM, the SAX parser does not allow random dynamic access to the XML document. But it does have an advantage predominantly in the parsing of large documents. A SAX parser is more memory efficient, since it does not need to store the entire document in memory and only uses the part of the document needed at specific times. As a result, a SAX parser could parse a document even larger than the memory of the system. The downside of SAX is that one must implement all possible event handlers to handle any event that may be encountered when parsing an XML document. Since there doesn't exist a tree structure like DOM, the application has to maintain the event state and keep track of where the parser is in the document hierarchy. Therefore with larger and more complex XML documents, the application logic for parsing the document can quickly become very complex despite the parser's memory-efficiency.

2.3 StAX Parser

StAX is a more recent stream-oriented parser. Like SAX it uses an event-driven model. However, instead of using a push model and pushing events to the event handlers, StAX uses a pull model to process events. StAX does not use a call back style like SAX. Instead StAX returns events when requested by the application. Unlike DOM and SAX, StAX has two parsing models: cursor model and iterator model. Like SAX the cursor model returns events. As a cursor can be moved through text in a text editor, StAX will pass through the document and execute certain tasks when encountering specific events. The iterator model returns each event as an object similar to those in a DOM tree. This model makes for a simpler interface, but with the additional overhead of creating objects. If using the cursor model, one advantage of StAX over SAX would be that the application controls the parsing process. This will make the code simpler to write and maintain regardless of the size of the XML document. If using the iterator model, one does not run into the complication of maintaining state and location of the parser within an XML document, since StAX only returns events explicitly requested by the application. StAX though does not have navigational support like DOM.

Table 1 describes the major parsers we studied to better understand these parsing techniques. Popular as they are, none really addressed the simplicity we aimed for in building an error-tolerant XML parser.

Parsers	**Language**	**Technique**	**DTD Validation**	**Schema Validation**	**Read/ Write XML**	**URL**
Sun SAX Parser	Java	SAX	yes	no	read	http://www.doc.ic.ac.uk/~sjn5/xml-tr2/readme.html
Apache XercesJ	Java	SAX	yes	yes	read	http://xml.apache.org/xerces2-j/index.html
Apache Xerces	C++	SAX /DOM	yes	yes	read	http://xml.apache.org/xerces-c/index.html
IBM Alpha Works	Java	SAX	yes	yes	both	http://www.alphaworks.ibm.com/tech/xml4j
IBM Alpha Works	C++	SAX /DOM	yes	yes	both	http://www.alphaworks.ibm.com/tech/xml4c
Expat	C	StAX	no	no	both	http://expat.sourceforge.net

Table 1 Characteristics of major parsers

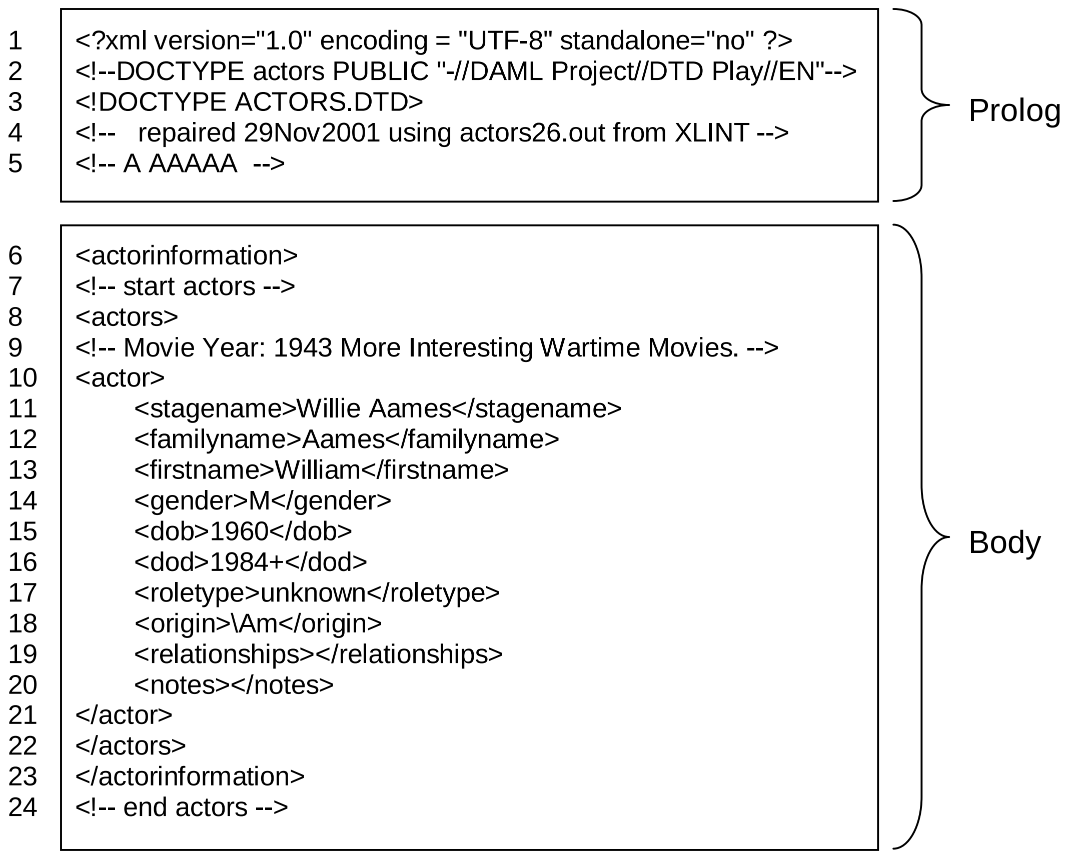
3 Approach

A previous effort resulted in a version of Xlint which is built upon the event-based commercial parser called Ælfred [3]. The approach was to override the Ælfred parser's error reporting mechanisms so that it can react to errors differently. Because of the difficulty in gaining control over the parsing process, the resulting parser cannot fully tolerate all the errors. It still fails in many cases before completing a single parse of the entire document.[†]

The approach we have taken in this new version of Xlint is to create the parser from scratch so that we have the full control over the document segment under inspection, the errors encountered and the recovery procedure from these errors.

3.1 XML Document Structure

Before we describe the basic algorithm, we first give an example of a well-formed XML document and a description of its structure defined according to the XML specification [1].



```
<?xml version="1.0" encoding = "UTF-8" standalone="no" ?>
<!--DOCTYPE actors PUBLIC "-//DAML Project//DTD Play//EN"-->
<!DOCTYPE ACTORS.DTD>
<!--   repaired 29Nov2001 using actors26.out from XLINT -->
<!-- A AAAAA  -->
```

```
<actorinformation>
<!-- start actors -->
<actors>
<!-- Movie Year: 1943 More Interesting Wartime Movies. -->
<actor>
        <stagename>Willie Aames</stagename>
        <familyname>Aames</familyname>
        <firstname>William</firstname>
        <gender>M</gender>
        <dob>1960</dob>
        <dod>1984+</dod>
        <roletype>unknown</roletype>
        <origin>\Am</origin>
        <relationships></relationships>
        <notes></notes>
</actor>
</actors>
</actorinformation>
<!-- end actors -->
```

Figure 1. A well-formed XML document

[†] The exact reason is unknown (probably even to the author himself due to the passing of time) since the documentation is vague and incomplete.

An excerpt from the actor XML file in the movie database is shown in Figure 1. An XML document contains a *prolog* and a *body*, where the prolog contains:

- an optional XML declaration at the beginning (line 1);
- any number of comments enclosed by "<!--" and "--!>" (line 2, 4, 5);
- any number of processing instructions enclosed by "<?" and "?>" (none in this example);
- an optional document type declaration (line 3).

And the body contains:

- any number of nested tagged text data (line 6, 8, 10-23);
- any number of comments enclosed by "<!--" and "--!>" (line 7, 9, 24);
- any number of processing instructions enclosed by "<?" and "?>" (none in this example).

3.2 Error Types

Based on this document structure, we classify the errors in a non-well-formed XML document into the following two types:

(1) Syntax errors, including:

- Syntax error for the xml declaration;
- Expect the attribute for version info;
- Invalid version number assignment;
- Invalid encoding name assignment;
- Invalid assignment for standalone document declaration;
- The specified attribute was not expected at this location;
- Syntax error for the tag (such as missing the end bracket);
- Duplicate DocType declaration;
- Syntax error for the comment;
- Syntax error for the processing instruction;
- Expect white space;
- Syntax error for attribute value assignment.

(2) Structural errors, including:

- Missing the start tag;
- Missing the end tag.

3.3 The Parsing Algorithm

The parser parses the document in a recursive descent manner. However, instead of removing left-recursion in the grammar and building an action table to create such a parser, we build a parser directly from the grammar in the XML specification [1]. This is because the grammar of an XML document is quite simple, as is described in section 2.1.

The parsing algorithm consists of two parts:

(1) Parse the prolog:

The parser starts by parsing the prolog of the document, branching to different handlers for declarations, comments and processing instructions, after recognizing each of them by their beginning characters. It reports syntax errors whenever it detects them. For example, when the following line is being parsed:

```
<?xml version="1.0" encoding = "UTF-8" standalon="no" ?>
```

The parser first recognizes this is an XML declaration by the first few characters - “<?xml”. It then checks all the optional and required attributes. In this example, it detects and reports the error that the attribute name “standalon” is incorrect. During the parsing of prolog, the parser only detects and handles the first type of errors, i.e, syntax errors.

(2) Parse the body:

When the first start-tag is detected (i.e., a tag which is not a declaration, comment or processing instruction), the parser begins to parse the document body. As opposed to parsing the prolog, the parser needs to detect structural errors (type 2 errors) as well as syntax errors for the document body. An example of structural error is shown in Figure 2.

```
… …
… …

<A>
        <B>
        <!-- comment 1. -->
                <C>
                        <D>text 1           // missing end tag </D>
                        <E>text 2</E>
                </C>
        </B>
        <!-- comment 2 -->
</A>
… …
… …
```

Figure 2. An example of structural error

The structural errors are detected and handled using a *tag-stack*. The basic algorithm is:

- For each empty-tag <content/>, ignore and keep parsing the rest of the document;
- For each start-tag <content> encountered, push “content” onto the tag-stack;
- For each end-tag </content> encountered, look up the stack for a matching tag, starting from the top element down to the bottom of the stack.

- If the top element matches the "content", we pop out the top element and no error occurs;
- If the matching element is not the top element, suppose its index in the stack is k, we pop out elements starting from the top element on the stack down to the (k+1)th element. For each element "tag-name" popped out, we report error as "missing the end-tag for tag-name". We then pop out the kth element (without reporting error).
- Otherwise, no matching tag is found. We report error as missing the start-tag for the tag under being parsed.

• When the entire document is parsed, if the stack is not empty, we clean up the stack by reporting "missing end-tag" error for each element on the stack.

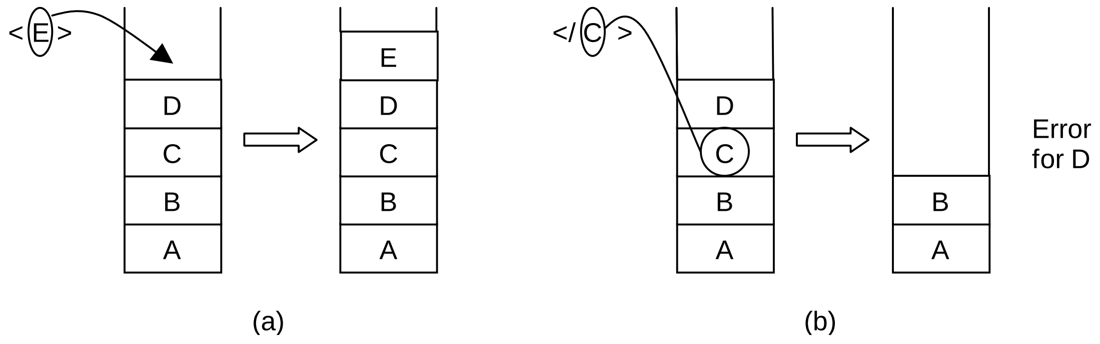


Figure 3. The tag-stacks for structural errors

This simple algorithm works for all the major structural errors. For example, in Figure 2, when the parser recognizes the start-tag <E>, it pushes the content E onto the tag-stack (see Figure 3(a)). Later, it pops the tag E from the stack after recognizing the end-tag </E>. Since the matching tag E is on top of the stack, no error occurs. Next, the parser gets the end-tag </C>. It examines the stack and discovers that the matching tag is not the top element, so it first pops D and reports an error as "missing end-tag for D"; then pops C as well but without reporting an error (see Figure 3(b)).

This algorithm also works for typos. Suppose we have </DD> instead of </D>, the </DD> will be ignored since it does not match any tags on the stack, then later the <D> will be popped out with an error of "missing end-tag for D". The error of an eager closing tag can also be captured. Suppose we have "…<A> <B> … </A>… </B>", the <B> will be popped out with an error of "missing end-tag for B" when the parsers gets the </A>; the </B> will be ignored later, since the <B> has already been popped out. Although a more accurate error description might be "incorrect nesting of tags" instead of "missing end-tag", we feel being able to do that may complicate the parser design.

Note that the document shown in Figure 2 is a simplified example because there are no attributes used by any of the entity elements. In the general case, when encountering a start-tag containing attribute-value pairs, the parser will first extract the tag-name for testing structural errors as is described above; the rest of the attribute-value pairs are parsed separately to check for syntax errors. For example, suppose the parser receives a start-tag <A name=”Mike” age=”12”>, it first extracts “A” and pushes it onto the tag-stack, then parses “name=”Mike” age=”12”” to check for syntax errors.

4 Implementation

Xlint is implemented using Perl. Perl provides a powerful string manipulation facility using regular expressions. This greatly facilitates the detection of different constructs of the document such as declarations, comments and tags. The difficult part is we need to be careful about the order of checking different constructs, because the pattern matching follows a greedy process in Perl. For example, when checking syntax errors within a list of attribute-value assignments (contained in a start-tag), we need to work backwards – starting from the last assignment at the end of the assignment list.

The document is parsed line by line, with blank lines skipped. The current line being parsed is stored in two variables, a *line* variable which keeps getting chopped off as the parser moves on; and a *cachedLine* variable which stays the same until the current line is completely parsed; it is used to provide context for error reporting. A *position* variable is used to track which column we have reached during the parsing on the current line (also used for error reporting).

We implemented the stack data structure providing parser-specific functions such as *locateTagname(tagname)*, which returns the depth of the matching tag by searching from the top of the stack; and *popFrom(d)*, which pops elements starting from the top element to the d-th one and reports error for each of them. Each element on the stack is a four tuple <tag-name, line-number, position-number, cached-line>. Line-number and position-number is used for pinpointing the error. The cached-line is used for providing the error context if an error is reported for the corresponding tag-name (whose associated line may not be the current one).

5 Experiments

We tested the parser on a number of movie files. The result is shown in Table 2.

XML document	Size (Byte)	Number of Errors
mains106.xml	5,532,867	2772
casts101.xml	5,060,207	794
actors51.xml	2,439,841	346
people37.xml	1,109,759	467

Table 2. The results of parsing large XML documents

6 User commands

The Xlint is executed by the command:

```
perl xlint.pl <file_name> [-v |-v <number_of_chars>]
```

The user must supply the "file_name" parameter which is the absolute or relative file name of the XML document to be parsed. Followed by the "file_name" is the optional parameters "-v" or "-v number_of_chars":

-v	The verbose mode with default context length. A context of 30 characters around the error position is displayed.
-v number_of_chars	The verbose mode with given context length. The length of error context is set to number_of_chars.

References:

[1] Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000, http://www.w3.org/TR/2000/REC-xml-20001006

[2] Gio Wiederhold, Movies Database Documentation, http://kdd.ics.uci.edu/databases/movies/doc.html

[3] Vincent Chu, The Report on Xlint.