A Table-Driven Streaming XML Parsing Methodology for High-Performance Web Services

Wei Zhang & *Robert van Engelen*[*]
Department of Computer Science and School of Computational Science
Florida State University, Tallahassee, FL 32306-4530
{wzhang,engelen}@cs.fsu.edu

Abstract

This paper presents a table-driven streaming XML parsing methodology, called TDX. TDX expedites XML parsing by pre-recording the states of an XML parser in tabular form and by utilizing an efficient runtime streaming parsing engine based on a push-down automaton. The parsing tables are automatically produced from the XML schemas of a WSDL service description. Because the schema constraints are pre-encoded in a parsing table, the approach effectively implements a schema-specific XML parsing technique that combines parsing and validation into a single pass. This significantly increases the performance of XML Web services, which results in better response time and may reduce the impact of the flash-crowd effect. To implement TDX, we developed a parser construction toolkit to automatically construct parsers in C code from WSDLs and XML schemas. We applied the toolkit to an example Web services application and measured the raw performance compared to popular high-performance parsers written in C/C++, such as eXpat, gSOAP, and Xerces. The performance results show that TDX can be an order of magnitude faster.

1. Introduction

XML validation, as performed by a *validating XML parser*, filters valid XML from invalid content based on a *schema*, e.g., XML schema [19], Relax-NG [5], or DTD. In the SOAP/XML Web services context where XML parsing is an essential component for message exchange, this simple principle guides the separation of concerns between message content verification and the service application logic.

Despite the key advantage of XML validation, it is often disabled for high-performance XML applications because it is well known that validation incurs significant processing overhead in the XML parser at the receiving server [7, 13]. Part of the overhead is caused by the checking of validation constraints in a separate stage after document parsing as described by the specifications [5,19]. A validating parser must keep sufficient state information, or even the entire document in memory, to validate an inbound XML stream against a schema. Furthermore, the complexity of the validation process is exacerbated when streaming XML parsing requirements are considered, due to the inherent incompatibility of streaming with a staged validation process. Streaming techniques allow XML messages to be parsed and processed on the fly, eliminating the need to store a document in full, such as with DOM.

The logical separation of parsing and validation in XML parsers and the frequent access to schemas can be eliminated by integrating parsing and validation into a *schema-specific parser* [3, 4, 13, 14, 16, 17]. Schema-specific parsers encode parsing states and validation rules to boost performance. Previous work [3,4,7,13,16] in this area was limited to DTDs or a limited subset of XML schema and does not include namespace support which makes these approaches useless for XML Web services. Prior to that work, the *gSOAP toolkit project* [14, 15, 17, 18] implemented schema-specific parsing based on *recursive descent parsers* [1] generated from XML schema via intermediate representations. Based on this experience with *gSOAP*, we developed TDX toolkit for streaming XML parsing as a faster, more economical method, to implement high-performance parsers. TDX is a parsing methodology that provides a flexible framework that combines well-formedness parsing, type-checking validation, and application-specific events.

The TDX methodology is based on linguistic principles by encoding XML parsing and validation inseparably in *LL(1) context-free grammar rules* [1, 7]. All structural constraints, many types of validation constraints (such as `minOccurs`, `maxOccurs`, `length`, `enumeration`, `boolean` values, etc.), imposed by XML schema, are incorporated in grammar productions. Checking of these constraints proceeds automatically through the grammar pro-

* Supported in part by NSF grant BDI-0446224, and DOE Early Career Principal Investigator grant DEFG02-02ER25543

ductions at run time. Other type constraints and application-specific events are encoded as semantic actions in the augmented grammar. Because validation is tightly integrated into the production rules, validation does not need to be enforced separately.

The semantic actions use index tokens that refer to application functions. The application functions are implemented in a "action table" in the service back-end. Therefore, TDX offers a high level of modularity, because the use of index does not require the re-compilation of application when the modular parse tables are generated for new or updated schemas.

The remainder of this paper is organized as follows. We first review related work in Section 2. Then, we describe TDX architecture in Section 3. Section 4 describes the table-driven XML streaming parsing. The mapping from XML schema to an augmented LL(1) grammar is given in section 5. Section 6 describes the automatic generation of a TDX parser with our code generator. Performance results are presented in Section 7. Finally, we summarize the paper with conclusions in Section 8.

2. Related Work

Van Engelen in [16] presented a method to integrate parsing and validation into a single stage with parsing actions encoded by a *deterministic finite state automaton* (DFA), where the DFA is directly constructed from a schema based on a set of mapping rules. DFA parsing is fast and combines parsing and validity checks. However, because of the limitations of the regular language described by DFAs, the approach can only be used to process a non-cyclic subset of XML schemas.

Chiu et al. [3, 4] also suggests an approach to merge all aspects of low-level parsing and validation by constructing a single *push-down automaton*. However, their approach does not support XML namespaces, which is essential for SOAP compliance. Furthermore, the approach requires conversion from a non-deterministic automaton (NFA) to a DFA. This conversion may result in exponentially growing space requirement caused by *subset construction* [1] or *powerset construction* [11].

In earlier work on the *gSOAP toolkit* [14, 17] a schema-specific parsing approach was implemented and a compiler tool was developed to generate LL(1) recursive descent parsers to efficiently parse XML documents with namespaces defined by schemas. This approach has the disadvantages of recursive descent parsing, which include code size and function calling overhead.

Also, the *Packrat parser* [6] implements recursive-descent parsing with backtracking and has been applied to XML. It achieves linear time parsing, but the main disadvantage is its space requirement, which is directly proportional to the XML input size. This rules *Packrat* out as a practical streaming XML parser, because the XML document length may, in practice, be quite large.

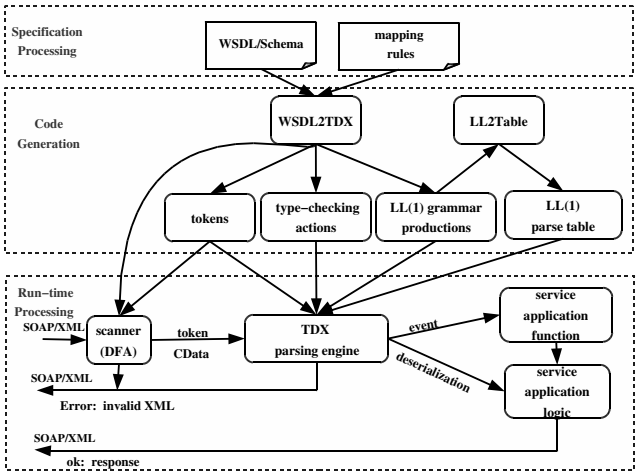

Figure 1: Architecture overview of TDX

Tree grammars were developed to represent the structure of XML instances [9, 10]. Tree grammars provide an efficient way to encode schema constraints, but they are not suitable to integrate parsing and validation into one stage.

3. Overview of TDX

The TDX methodology consists of three stages: specification processing, code generation, and run-time processing. Figure 1 depicts the architecture overview of TDX.

TDX uses mapping rules to translate WSDL or XML schema description into LL(1) grammar productions. The mapping rules define the mapping from schema components to LL(l) grammars.

Code generation requires two code generators: *WSDL2TDX* and *LL2Table*. *WSDL2TDX* takes the WSDL or Schema description and the mapping rules as input, produces source code automatically for DFA-based scanner, tokens, type-checking actions, and LL(1) grammar productions. The *LL2Table* then constructs an LL(1) parse table from generated LL(1) grammar. The application-specific functions cannot be generated automatically because they are application-dependent. The application-specific actions must be manually inserted before the LL(1) parse table is constructed.

At run time, the scanner scans the XML stream and breaks it up into a token stream to the TDX parsing engine. The engine performs well-formedness verification, type-checking validation, generates events to invoke application-specific actions, and deserializes data to the application's back-end. Syntactical and validation errors are returned to the client as SOAP faults. Validation errors indicate schema constraint violation.

4. Table-Driven XML Processing

The table-driven parsing technique eliminates the overhead generated by recursive and backtracking calls, and its space requirement is linear to the nesting depth of XML document. These speed and memory-efficient advantages make the table-driven parsing technique amenable for constructing streaming XML processors.

4.1. Tokenizing XML

By tokenizing XML into tokens, we enhance the overall performance, because matching tokens once is more efficient than repeatedly comparing strings in the parser. Tokens are defined by schema component tags such as element names, attribute names. In our approach we namespace-normalize XML tags and then classify the XML tag names according to uses. All beginning element tags `<NAME>` are represented by `bNAME`, ending element names `</NAME>` are represented by `eNAME`, and attribute tag names are represented by `aNAME`. Similarly, enumeration values `value='V` are represented by `cV`. Enumeration values are also defined as tokens, mainly to improve performance.

Namespace bindings are supported by internal normalization of the token stream to simplify the construction of LL(1) parse table. Namespace qualified elements and attributes are translated into normalized tokens according to a namespace mapping table. Thus, identical tag names defined under two different namespace domains are in fact separate tokens. Table 1 lists tokens for an example schema.

4.2. Annotated LL Grammar Productions

The LL(1) grammar productions are generated from a schema using *WSDL2TDX*. Application-specific actions can be semi-automatically or manually inserted to fire events to the application. Each production contains a nonterminal at its left-hand side of '→', followed by a sequence of nonterminals and terminals at its right-hand side. Two special tokens `CDATA` and `EVENT` are used as indicators to invoke the type-checking actions for content validation, and to generate events to invoke application-specific functions respectively. Suppose the application logic has a function processing the element *state* when the state is `'ON'`, and we also assume the function is indexed as *i* in the action table for the event dispatcher, then the table 2 shows the generated LL(1) grammar for the schema in Table 1.

Token	XML Schema
	<schema>
	<simpleType name="state_type">
	<restriction base=""xsd:string>
cOFF	<enumeration value="OFF"/ >
cON	<enumeration value="ON"/ >
	< /restriction>
	< /simpleType>
	<complexType name="example_type">
	<sequence>
bID	<element name="id" type="xsd:string"
eID	/ >
bVALUE	<element name="value" type="xsd:integer"
eVALUE	/ >
bSTATE	<element name="state" type="state_type"
eSTATE	/ >
	<sequence>
	< /complexType>
	< /schema>

Table 1: Tokens for an example schema

#	Production	Action
1	$s \rightarrow$ cOFF	
2	$s \rightarrow$ cON EVENT	gen_event(*i*)
3	$t \rightarrow t_1 t_2 t_3$	
4	$t_1 \rightarrow$ bID CDATA eID	imp_string(s.*val*)
5	$t_2 \rightarrow$ bVALUE CDATA eVALUE	imp_integer(s.*val*)
6	$t_3 \rightarrow$ bSTATE s eSTATE	

Table 2: LL(1) grammar generated from Schema in Table 1

4.3. LL Parse Table

The LL(1) parse table is constructed from the LL(1) grammar productions using *LL2Table*. The LL(1) grammars are unambiguous and free of left recursion. Given a nonterminal `X` and terminal (i.e. XML token) `a`, the pair `<X,a>` uniquely determines the production to be selected at run time, for constructing a parse tree. This property allows us to construct a parse table for the TDX parsing engine to get the appropriate production.

4.4. Type-Checking Actions

The table with type-checking actions contains the actions for type validation as imposed by XML schema. Although many structural validation rules can be cast as grammar productions, others have to be validated explicitly with semantic actions. Those include most `XSD` types such as `float`, and `integer`, and most `simpleType` defined by `restriction`.

4.5. Scanner

The scanner is the runtime tokenizer that scans the XML stream and generates a token stream for the TDX parsing engine. The scanner is generated from WSDL or schema description as described in Section 4.1, thus specialized to the XML streaming message to efficiently perform the lexical analysis. It produces a token stream using a DFA-based token recognizer that is automatically produced with Flex [8] at compile time.

4.6. TDX Parsing Engine

The TDX parsing engine is schema- and application-independent. It consumes the scanner's token stream and performs the XML structure verification, type-checking validation, and invokes application actions, by consulting the TDX tables. It also deserializes the data for the application logic. The XML well-formedness is verified through the grammar productions. Type-checking validation is performed using actions after special token `CDATA` has been seen. The application-specific actions are invoked when the special token `EVENT` has been met. TDX parsing engine uses a predictive top-down parsing method with one token lookahead. It avoids the overhead of recursive and backtracking methods, and its space requirement achieves efficiency that is linear to the nesting depth of XML elements.

The TDX parsing engine maintains a local stack to track the parser's states. The stack contains a sequence of grammar symbols with `$`, a symbol used as an end-marker, on the bottom, indicating the bottom of the stack. The engine implements an LL(1) predictive parsing algorithm[1] described in [1]. The token on top of the stack, `X`, and the current input token, `a`, determine the behavior of the engine:

a. If `X=a=$`, the engine announces success.

b. If `X=a=CDATA`$\neq$`$`, the engine pops `X`, invokes the corresponding function to perform type-checking validation, then reads next input token on successful validation, halts and announces a validation error otherwise.

c. If `X=a=EVENT`$\neq$`$`, the engine pops `X`, generates events to invoke the corresponding application-specific function, deserializes data to the application, then reads next input token upon completion of invocation and deserialization.

d. If `X=a`$\neq$`CDATA`$\neq$`EVENT`$\neq$`$`, the engine pops `X`, then the reads the next input token.

e. If `X` is a non-terminal, the engine consults entry `T[X][a]` of the parse table. The entry will be either an index of an `X`-production or an error entry. If it is an error entry, the engine halts and announces a syntax error. Otherwise, it gets the production using the index, replaces the `X` with its right side token(s) in reverse order, i.e., the rightmost token is on top of the stack.

5. Mapping XML Schema to an LL Grammar

In this section we describe the details of the mapping rules that define the translations from XML schema components to LL(1) grammar productions. The mapping preserves the structural and semantic constraints on XML instances of XML schemas.

5.1. Terms and Notations

We use X to represent an arbitrary closed schema component. We also use x to represent a substring in a schema component without occurrence or "`use`" constraint. The symbol N denotes a non-terminal.

The term $[N]_m$ is used to represent a string of N's with exact m number of occurrences of N's.

The mapping operator $\Gamma[\![X]\!]N$ takes a schema component and a designated non-terminal N, and returns a set of LL(1) grammar productions. The symbol N', N'', and N_i represent new nonterminals derived from nonterminal N.

5.2. Mapping Rules

Table 3 shows a subset of the mapping rules for schema constraints. Other components such as `complexType` and top-level `element` and `attribute` definitions are mapped in a similar way, see [20] for more details. Mapping operations are defined recursively. For example, rule (1) takes a schema component x with an occurrence constraint and a nonterminal N. It returns an epsilon production and a mapping operation to produce the remaining productions for x. Occurrence constraints `minOccurs` and `maxOccurs` determine the minimum and maximum occurrences of an element or other component. Occurrence constraints are mapped to the `epsilon` production for `minOccurs='0'` (rules (1)–(2)) and right recursive production for `maxOccurs='unbounded'` (rules (2)–(4)). Rule (5) also defines mapping for occurrence constraints to preserve that the component $< x/>$ can appear m to n times. Rule (6) maps occurrence constraint `use='optional'`, e.g. for attributes, to the `epsilon` production.

Ordering constraints, such as `all`, `choice`, and `sequence`, which are common particles of the `complexType` component, determine in what order elements should occur. Rules (8)–(10) define mappings for these order constraints. The `all` particle requires that the child elements can appear in any order and that each child element must occur once and only once. The grammar production $N \rightarrow A(N_1, N_2, \ldots, N_n)$ in rule (8) repsents that the terminals N_1, N_2, ..., and N_n can appear in any order but must appear one and only once. This special grammar production can be implemented using a stack or a flag. The `choice` specifies that either one child element or another can occur. The `sequence` specifies that the child elements must appear in the specific order. Rules (11-12) define mappings for `group`.

1 Some schema may, although rarely, have very large number of occurrence, say `minOccurs="5000"`. In that case, this algorithm may be inefficient in terms of memory requirements.

Rule#	Translation		
1	$\Gamma[\![< x \text{ minOccurs='0'/>}]\!]N$	$=$	$\{N \to \epsilon\} \cup \Gamma[\![< x/>]\!]N$
2	$\Gamma[\![< x \text{ minOccurs='0' maxOccurs='unbounded'/>}]\!]N$	$=$	$\{N \to N'N, N \to \epsilon\} \cup \bigcup \Gamma[\![< x/>]\!]N'$
3	$\Gamma[\![< x \text{ maxOccurs='unbounded'/>}]\!]N$	$=$	$\{N \to N'N'', N'' \to N'N'', N'' \to \epsilon\} \cup \Gamma[\![< x/>]\!]N'$
4	$\Gamma[\![< x \text{ minOccurs='}m\text{' maxOccurs='unbounded'/>}]\!]N$	$=$	$\{N \to [N']_m N'', N'' \to N'N'', N'' \to \epsilon\} \cup \Gamma[\![< x/>]\!]N'$
5	$\Gamma[\![< x \text{ minOccurs='}m\text{' maxOccurs='}n\text{'/>}]\!]N$	$=$	$\{N \to [N']_m N'_1, N'_1 \to N'N'_2, N'_2 \to N'N'_3, \ldots, N'_1 \to \epsilon, \ldots\} \cup \Gamma[\![< x/>]\!]N'$
6	$\Gamma[\![< x \text{ use='optional'/>}]\!]N$	$=$	$\{N \to \epsilon\} \cup \Gamma[\![< x/>]\!]N$
7	$\Gamma[\![< x \text{ use='required'/>}]\!]N$	$=$	$\Gamma[\![< x/>]\!]N$
8	$\Gamma[\![<\text{all}> X_1 X_2 \ldots X_n <\text{/all}>]\!]N$	$=$	$\{N \to A(N_1, N_2, \ldots, N_n)\} \cup \bigcup_{i=1}^{n} \Gamma[\![X_i]\!]N_i$
9	$\Gamma[\![<\text{choice}> X_1 X_2 \ldots X_n <\text{/choice}>]\!]N$	$=$	$\bigcup_{i=1}^{n} (N \to N_i) \cup \bigcup_{i=1}^{n} \Gamma[\![X_i]\!]N_i$
10	$\Gamma[\![<\text{sequence}> X_1 X_2 \ldots X_n <\text{/sequence}>]\!]N$	$=$	$\{N \to N_1 N'_1, N'_1 \to N_2 N'_2, \ldots, N'_{n-1} \to N\} \cup \bigcup_{i=1}^{n} \Gamma[\![X_i]\!]N_i$
11	$\Gamma[\![<\text{group name='G'}> X <\text{/group}>]\!]N$	$=$	$\Gamma[\![X]\!]G$
12	$\Gamma[\![<\text{group ref='G'/>}]\!]N$	$=$	$\{N \to G\}$

Table 3: Mapping occurrence, order and group constraints

The following example illustrates the mapping process:

$\Gamma[\![$<complexType name='C'><sequence minOccurs='0'>
<element name='A'type='AT'/><element name='B' type='BT'/>
</sequence></complexType>$]\!]N$
$= \Gamma[\![$<sequence minOccurs='0'><element name='A' type='AT'/>
<element name='B' type='BT'/></sequence>$]\!]C$
$= \{C \to \epsilon\} \cup \Gamma[\![$<sequence><element name='A' type='AT'/>
<element name='B' type='BT'/></sequence>$]\!]C$
$= \{C \to \epsilon, C \to C_1 C'_1, C'_1 \to C_2\}$
$\cup \Gamma[\![$<element name='A' type='AT'/>$]\!]C_1$
$\cup \Gamma[\![$<element name='B' type='BT'/>$]\!]C_2$
$= \{C \to \epsilon, C \to C_1 C'_1, C'_1 \to C_2,$
$C_1 \to$ bA AT eA, $C_2 \to$ bB BT eB$\}$

5.3. Dealing with Non-LL Grammar Properties

It is critical to preserve the LL(1) property for constructing the LL(1) parsing table, otherwise the TDX parser construction fails. A few special cases, usually involving the `choice` particle and `all` particle, may end up producing a non-LL(1) grammar due to ambiguity in the schema.

Consider for example the schema fragment shown in Table 4. The `choice` has two child elements with the same name `"A"`. This generates a non-LL(1) grammar shown in Table 4. Note that both of the two productions with the same nonterminal, $C \to C_1$ and $C \to C_2$, can be used to generate element `"A"`. This results in a parsing conflicts when constructing the LL(1) parse table, i.e., either one of the productions ends up at the same place in the parse table.

The code generator *WSDL2TDX* uses *left factoring* to eliminate conflicts [1]. To be at the safe side, the generator also performs *left recursion* check to make sure the generated grammar is LL(1).

Note that other particles such as `sequence`, `all`, and `any`, do not allow their child elements to have the same name under the same namespace. As result, such particles do not result in parsing conflict arising from ambiguities caused by `choice`.

Schema Fragment	Non-LL(1) Grammar	LL(1) Grammar

`<complexType name="C">`
`  <choice>`
`  <sequence>`
`  <element name="A"`
`  type="T"/>`
`  <element name="B"`
`  type="T"/>`
`  </sequence>`
`  <element name="A"`
`  type="T"/>`
`  </choice>`
`</complexType>`
	$C \to C_1$
	$C \to C_2$
	$C_1 \to C_3 C'_3$
	$C'_3 \to C_4$
	$C_3 \to$ bA T eA
	$C_4 \to$ bB T eB
	$C_2 \to$ bA T eA
		$C \to C_1 C'_1$
		$C_1 \to$ bA T eA
		$C'_1 \to$ bB T eB
		$C'_1 \to \epsilon$

Table 4: Schema fragment generating non-LL(1) grammar

6. Code Generator Implementation

This section describes the code generation implementation details for automatic construction of TDX parsers.

6.1. Token Generator Implementation

The TDX code generator *WSDL2TDX* scans WSDL or schema description and generates tokens for each element name, attribute name and enumeration value by assigning a unique number to each token. Each token is defined by a tag name under a specific XML namespace. A default namespace "" is assigned for a tag name that has no explicit targetNamespace.

Namespace URIs and tokens are stored in a 1-D array `ns[NS_SIZE]` and a 2-D array indexed by namespace and name `tok[NS_SIZE][NAME_SIZE]`, respectively. Both arrays are used to generate tokens for the grammar productions in the table. The generated scanner also consults both arrays to break input stream into tokens. Each tag name is defined as macro. The code generator generates ANSI C codes for the tag names and the namespaces from the schemas. For example, the fragment of source code generated from the schema shown in Table 1 is:

```
#define ON        0
#define OFF       1
#define ID        2
#define VALUE     3
#define STATE     4
...
tok[0][ON]    = 100
tok[0][OFF]   = 102
tok[0][ID]    = 104
tok[0][VALUE] = 106
tok[0][STATE] = 108
```

Here we assume the default namespace is stored in `ns[0]`. Tokens are stored only once to minimize memory use. A token with an even number denotes the token of a beginning element tag. The token of an ending element is represented as the next number (odd) of its corresponding beginning token. For example, `bVALUE = 106` and `eVALUE = 107`.

6.2. Scanner Generator Implementation

A Flex description is generated by *WSDL2TDX*. The scanner is based on the content of the WSDL or schema description. Flex is a frequently used automatic generator tool by compiler developers for high-performance DFA-based scanners [1, 8]. The generated Flex specification is fed into Flex to produce the ANSI C source code for the XML scanner. This Flex description scans an input stream and breaks it into strings which match the given expressions. The recognition of the expressions is performed by the code generated by Flex. The generated Flex description has the following structure[2]:

```
1  %{ ... %}
2  whsp [ \t\v\n\f\r]
3  name    [^>/:= \t\v\n\f\r]+
4  qual    {name}:
5  open    <
6  stop    >
7  skip    [^/>]*
8  data    [^<]*
9  xmln xmlns(:{name}|"")=(\"[^"]*\"|\'[^']*\')
10 attr    =(\"[^"]*\"|\'[^']*\')
11 %x OPEN_ELEM
12 %x CLOS_ELEM
13 %x ATTS
14 %x CDATA
15 %x PC_ENUM
16 %%
17 {whsp}                // ignore white space
18 {open}"?"{skip}{stop} // ignore declaration
19 {open}"!"{skip}{stop} // ignore comment
20 {open}"/"             {RESETNS; BEGIN(CLOS_ELEM);}
21 {open}                {RESETNS; BEGIN(OPEN_ELEM);}
22 <OPEN_ELEM>{whsp}     // ignore white space
23 <OPEN_ELEM>{qual}     ns = get_ns(yytext);
24 // ... definitions of XML element names and actions
25 <CDATA>{data}         {return CDATA; BEGIN(INITIAL);}
26 <CLOS_ELEM>{whsp}     // ignore white space
27 <CLOS_ELEM>{qual}     ns = get_ns(yytext);
28 // ... definitions of XML element names and actions
29 <CLOS_ELEM>{stop}     BEGIN(INITIAL);
30 <ATTS>{whsp}          // ignore white space
31 <ATTS>{qual}          ns = get_ns(yytext);
32 // ... definitions of XML attribute name and actions
33 <ATTS>xmln            ins_ns(yytext);
34 <ATTS>{stop}          BEGIN(CDATA);
35 <*><<EOF>>            return EOF;
36 %%
37 // user code definition
```

The Flex specification consists of three sections: *definitions*, *rules*, and *user subroutines*, separated by a line with '*%%*'. The *definition* section contains *name* definitions used to simplify the scanner specification, and *start conditions* providing a mechanism for conditionally activating rules. The second section *rules* contains of a series of rules of the form:

```
pattern action
```

When the substring of input stream matches the pattern, a regular expression, the action is executed to process accordingly. The last section consists of *user subroutines*, where user functions can be defined. This Flex specification extracts all the element names, attributes and enumeration values and returns a token stream. In the *definitions* section, lines 2–10 define a set of names, and lines 11–15 define a set of starting conditions. The line 16 marks the beginning of the second section *rules*. Lines 17–19 define rules ignoring white space, declaration and comment respectively because no action is defined for these patterns. Line 20 defines a rule that the scanner will enter the starting condition `<CLOS_ELEM>` when a pattern '`</`' is recognized, while line 21 defines a rule that the scanner will enter into the starting condition `<OPEN_ELEM>` when the pattern '`<`' is recognized. Because names may or may not be qualified, the `qual` regular expression is used to extract the possible name space for each element name and attribute name (line 23). Therefore the variable `ns` must be reset before starting to match a name each time. The function `get_ns(char *)` returns an integer representing the namespace. The function `ins_ns(char *)` installs and caches the namespace. XML attributes are cached for efficiency reasons and to resolve namespace bindings, because the `xmlns` binding may occur after an attribute was scanned.

As can be seen from this example, the generated scanner is specialized to the XML schemas at hand to improve the efficiency, compared to the generic scanner. Schema-specific regular expressions and actions are added to the description. This includes all the element and attribute names found in the WSDL or schemas. These element definitions are collected from all parts of a set of related schemas, including top-level element schema components and local elements.

For example, consider the schema in Table 1, the following actions are added to the *rules* section of Flex specification:

```
<PC_ENUM>'OFF'      {return tok[ns][OFF];BEGIN(INITIAL);}
<PC_ENUM>'ON'       {return tok[ns][ON];BEGIN(INITIAL;}
<OPEN_ELEM>'id'     {return tok[ns][ID];BEGIN(ATTS);}
<CLOS_ELEM>'id'     {return tok[ns][ID]+1;}
<OPEN_ELEM>'value'  {return tok[ns][VALUE];BEGIN(ATTS);}
<CLOS_ELEM>'value'  {return tok[ns][VALUE]+1;}
<OPEN_ELEM>'state'  {return tok[ns][STATE];BEGIN(PC_ENUM);}
<CLOS_ELEM>'state'  {return tok[ns][STATE]+1;}
```

6.3. LL Parse Table Generator Implementation

As was discussed previously, the LL(1) grammar productions are produced by *WSDL2TDX* from the WSDL description or schemas using the mapping rules outlined in Section 5. The generated grammar productions are stored in a 2-D array `P[I][S]`, where `I` is the production index and `S` indicates the grammar symbols. Terminals and non-terminals are represented as positive and negative integers respectively. Epsilon is represented using zero.

The code generator *WSDL2TDX* also generates a set of functions performing the type-checking validation. The

2 This Flex specification is simplified for clarity.

function pointers are stored in an array `pt2Func[I]`, where `I` is the index of the production. This offers a way to invoke a function associated with a production. For example , the following commented sample below illustrates the generated code for performing type-checking validation in Table 2.

```
// type-definition: define a function pointer type
typedef int (*pt2Function)(char *);
...
// define a function pointer array
// holding function pointers
// and initialize with NULL
pt2Function pt2Func[NUM_Production] = {NULL};
...
// assign the function's address
pt2Func[4] = &impl_string;
pt2Func[5] = &impl_integer;
...
// calling a function using an index
pt2Fun[4](yytext);
// calling impl_string(char *)
pt2Fun[5](yytext);
// calling impl_integer(char *)
```

The parsing table is constructed by *LL2table* from the grammar productions. The parsing table is implemented as a 2-D array `T[A][a]`, where `A` is nonterminal, and `a` is a terminal or endmarker `$`. All entries are initialized with `ERROR_ENTRY`, a negative integer indicating error entries. The parsing table is implemented based on the generated LL(1) grammar using algorithm 4.4 in [1]. Any conflict indicates a non-LL(1) grammar.

7. Performance Results

We compared our TDX framework to *gSOAP* 2.7, *eXpat* 1.2, *Apache Xerces* for C++ 2.7.0 and a very fast DFA-based XML parser [16]. All code was compiled using gcc 3.2.2 (Xerces compiled with g++ 3.2.3) with option -O2 on a 2.4GHz P4 CPU and 512MB of main memory machine running Red Hat Linux 3.2.2-5.

The XML schema used for testing is shown in Figure 2. This schema defines an `echoString` message element containing a child element `input` of type XSD string.

The raw XML parsing performance was measured on memory-resident messages. Therefore network bandwidth and I/O latency are not measured. This allows us to compare the raw parsing speeds. The first run is discarded (warm-up) and the timing of thousands of runs was measured with `gettimeofday()`.

```
<schema targetNamespace="urn:echoString"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="echoString">
      <complexType>
         <sequence>
             <element name="input" type="xsd:string"/>
         </sequence>
      </complexType>
   </element>
</schema>
```

Figure 2: XML Schema for echoString request message

Figure 3 shows the elapsed time in microseconds (μs) of the TDX-based parser, the DFA-based parser, *eXpat*, *gSOAP*, and *Xerces*. The size of the input message is 1024 bytes. The performance of TDX-based parser with a scanner produced with `Flex -Cfa`, and the performance of DFA-based parser with scanner produced with `Flex -Cfa` were also compared. The `Cfa` options generate a faster DFA scanner but one that is larger in size.

The TDX-based parser combines parsing and validation in one stage. The total time of TDX-based parser include scanning time and validation time. The scanning time indicates the time spent on scanning the message and serializing the message into token stream. The non-optimized TDX-based parser takes only one-fifth of total time on validation, and the optimized TDX-based parser spends one-third of total time on validation, i.e., the TDX-based parser spends most of the time on scanning message.

Apache Xerces for C++ is a validating XML parser written in a portable subset of C++ [2]. It was setup with XML validation and namespaces support. All other options are turned off to achieve better performance. The *eXpat* parser [12] is a non-validating streaming XML parser. It is considered one of the fastest streaming XML parsers.

The performance of *gSOAP* shown has two parts, because *gSOAP*'s parser was not designed to be used as a stand-alone parser. The validation/decoding part indicates the validation time and the time spent on deserializing the message for application. Thus the total time includes parsing time, validation time and deserialization time.

The result shows that the optimized TDX-based parser is 17 to 18 times faster than *Xerces*, and the non-optimized TDX-based parser is 13 times faster than *Xerces*, in terms of the total time for scanning, parsing and validation.

The performance of the optimized validating TDX-based parser is approximately three times faster than non-validating *eXpat* parser with namespace support. The performance of the *gSOAP* is seven times slower than the TDX-based parser with `Flex -Cfa`. Note that performance of TDX-based parser is lower than the DFA-based parser, but the difference is very small (The DFA-based parser is about 14% faster than TDX-based parser).

Figure 4 shows the performance to parse message of a given size in log scale. The x-axis shows the number of elements in the message, and the y-axis shows the total time in microseconds (μs). The results show that *Xerces* has a larger initial delay time than others. The TDX-based parser is about three times faster than *eXpat*, and six times faster than *gSOAP*, seven to eight times faster than *Xerces*. The result also shows that the TDX-based parser is comparable to the DFA-based parser in terms of performance.

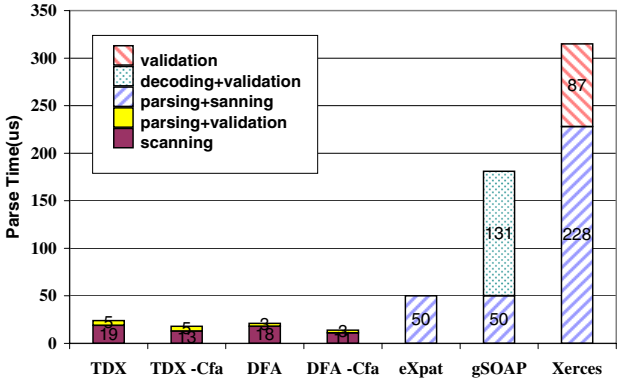

Figure 3: Performance of echoString parsing for n=1024 (2.40GHz P4)

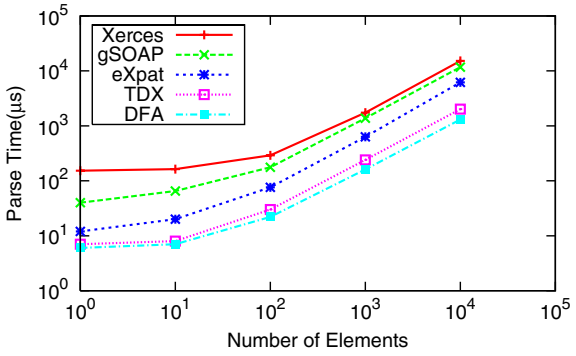

Figure 4: Performance of echoString parsing

8. Conclusion

This paper presented the TDX methodology for XML parsing, which utilizes a table-driven XML parsing approach. The TDX tables essentially implement a schema-specific XML representation for efficient parsing and validation. The presented approach has advantages for high-performance Web services. The results show that the speed of the parser is several times faster than industrial-strength high-performance validating XML parsers. The TDX approach also achieves a high level of modularity and adaptiveness for developing XML Web service applications, because TDX tables are interchangeable and can be easily replaced when a schema is updated, while purely compiled approaches require service applications to be recompiled.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading MA, 1985.

[2] Apache Foundation. Xerces XML Parser. Available from http://xerces.apache.org/.

[3] K. Chiu. Compiler-based approach to schema-specific XML parsing. Technical Report Computer Science Technical Report 592, Indiana University, 2003.

[4] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing*, pages 246–254, Edinburgh, Scotland, July 23-26 2002.

[5] J. Clark and M. Makoto. Relax NG specification, November 2001. Available from http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[6] B. Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology (MIT), September 2002.

[7] W. Lowe, M. Noga, and T. Gaul. Foundations of fast communication via XML. *Annals of Software Engineering*, 13:357–379, 2002.

[8] T. Mason and D. Brown. *Lex & Yacc*. O'Reilly and Associates, Inc., 632 Petaluma Ave, CA, 1990.

[9] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, 2001.

[10] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3), 2002.

[11] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, Boston, MA, 1997.

[12] SourceForge.net. The Expat XML Parser. Available from http://expat.sourceforge.net/.

[13] H. S. Thompson and R. Tobin. Using finite state automata to implement W3C XML schema content model validation and restriction checking. In *Proceedings of XML Europe*, 2003.

[14] R. van Engelen. The gSOAP toolkit 2.1, 2001. Available from http://gsoap2.sourceforge.net.

[15] R. van Engelen. Code generation techniques for developing light-weight efficient XML Web services for embedded devices. In *Proceedings of 9th ACM Symposium on Applied Computing SAC 2004*, pages 854–861, Nicosia, Cyprus, 2004.

[16] R. van Engelen. Constructing finite state automata for high performance XML web services. In *Proceedings of the International Symposium on Web Services (ISWS)*, 2004.

[17] R. van Engelen and K. Gallivan. The gSOAP toolkit for web services and peer-to-peer computing networks. In *Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid*, pages 128–135, Berlin, Germany, May 2002.

[18] R. van Engelen, G. Gupta, and S. Pant. Developing Web services for C and C++. *IEEE Internet Computing*, pages 53–61, March 2003.

[19] W3C. XML Specification, February 2004. Available from http://www.w3.org/TR/2004/REC-xml-20040204.

[20] W. Zhang and R. van Engelen. Mapping XML schema to LL(1) grammar. Technical report TR-060801, Department of Computer Science, Florida State University, 2006.