PAPER ID = 257 — Submitted to the IEEE International Conference on Web Services 2004.

Constructing Finite State Automata for High Performance Web Services

Robert A. van Engelen[*]
Department of Computer Science and School of Computational Science and Information Technology
Florida State University, Tallahassee, FL 32306-4530
engelen@cs.fsu.edu

Abstract

This paper presents a new XML parsing method based on deterministic finite state automata (DFA). A DFA generator is described that automatically translates XML Schemas to DFAs for efficient parsing of XML documents and SOAP/XML messages. The DFA-based parsing approach supports the implementation of high-performance Web services. Two example case studies are described and performance results are given.

1. Introduction

Several studies on the performance of SOAP [2, 4, 6, 8] reported findings that suggest that SOAP is inefficient for high-end data transport. This paper presents an efficient XML parsing method to improve Web services performance. A deterministic finite state automata (DFA) is constructed to effectively reduce computational requirements for XML parsing of SOAP/XML messages by encoding the XML parser's states as a DFA. The DFA is automatically derived by a code generator that takes a WSDL or set of Schemas as input and generates codes for an optimized Schema-specific SOAP/XML message parser.

This paper is organized as follows. Section 2 describes the DFA-based parser generator for XML. The generator takes a WSDL or Schema and generates source codes for the implementation of a high-performance Web service. Section 3 presents a classification of XML documents and Schemas. This classification discerns the framework for the presented parsing approach. Section 4 describes the construction of the DFA-based parser and Section 5 presents two examples case studies demonstrating the construction process. Section 6 presents the performance improvement by comparing parsing speeds. The paper is summarized in Section 7 with some concluding remarks and suggestions for improvements.

* Supported in part by NSF grants CCR-0105422, CCR-0208892, and DOE grant DEFG02-02ER25543.

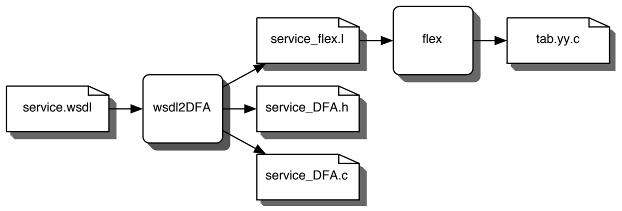


Figure 1. DFA-Based Parser Generator

2. A Parser Generator

Figure 1 depicts the DFA generator `wsdl2DFA`. The generator takes a WSDL description (or a set of Schemas) `service.wsdl` and produces a Flex description `service_flex.l` for the XML scanner and the DFA source code `service_DFA.c` for the XML parser. Flex is an automatic generator tool for high-performance scanners [1, 7]. Flex is mainly used by compiler writers to develop scanners that break up a character stream (program source code) into a sequence of tokens in the front-end of a compiler. The `service_flex.l` description generated by `wsdl2DFA` is fed into Flex to produce the source code for the XML scanner. This scanner provides a token stream to the XML parser at run time as shown in Figure 2. Both the generated scanner and the generated DFA-based parser are specialized to the SOAP/XML messages and Schemas defined in the WSDL to efficiently parse and decode Web service message invocations for a SOAP/XML Web service application.

The Web service application receives a sequence of

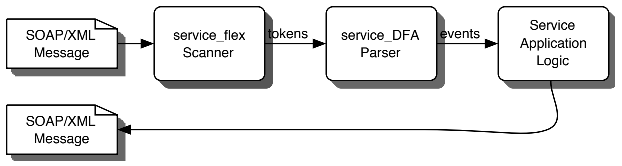


Figure 2. High-Performance Web Service

events from the parser corresponding to the parser's actions. This is somewhat similar to a SAX parser implementation. This event information includes XPath [15] descriptions and data containing the opening XML element tags (with attributes) or the content of elements with CDATA. The XPath descriptions indicate the XML source location of the elements and data parsed. The events are handled by the service application, which is left to the application developer to implement.

To generate a DFA-based parser for WSDL or a Schema, the Schema components must form a partially ordered set. The details of this requirement are given in the next section.

3. A Classification of XML Documents

In this section we introduce a classification of XML documents. The classification is intended to reflect common restrictions imposed on document structures encountered in practice. In addition, the classification provides a framework to formally define the requirements for XML parsing with DFAs.

The first obvious distinction we make in the classification shown in Table 1 is between arbitrary syntactically well-formed XML documents and documents that can be validated against a Schema or DTD. We further distinguish documents with cyclic ID-REF references (i.e. as per id/ref attributes declared as ID and REF Schema XSD types) from those that have acyclic references (Acyclic Type 2 documents), and those that have ordered, acyclic Schemas (Ordered Type 3 documents).

3.1. Acyclic Documents

An ID-REF cycle in an XML document or in a set of related XML documents is defined as follows.

Definition 3.1 *A (set of related) XML documents has an ID-REF cycle if an element E_1 has a REF attribute that matches the ID attribute of an element E_2, such that a descendant element of E_2 has a REF attribute that refers to an ancestor element of E_1. The descendant element of E_2 may be directly contained in the enveloping element E_2 or may be referenced through a forward pointing chain of ID-REF references originating from a descendant of E_2.*

Consider for example the following XML fragments:

```
<a>
  <b ref="#A"/>
  <c id="A">...</c>
  <d>...</d>
</a>
```

Acyclic

```
<a id="A">
  <b>
    <c ref="#A"/>
  </b>
  <d>...</d>
</a>
```

Cyclic

```
<a id="A">
  <b ref="#B"/>
</a>
<b id="B">
  <c ref="#C"/>
</b>
<c>
  <a ref="#A"/>
</c>
```

Cyclic

Type	Class	Description
0	Unrestricted	Any well-formed XML document.
1	Validated	Any valid XML document.
2	Acyclic	Documents free of ID-REF cycles.
3	Ordered	Document Schema components form a poset.

Table 1. XML Document Classification

The first example fragment has an acyclic ID-REF dependency between elements `b` and `c`. The second example is cyclic ($E_1 = E_2$ in Definition 3.1). The third example contains a cycle through a chain of ID-REF references.

Acyclic documents are very common, in contrast to documents with one or more cyclic references. Therefore, this restriction does not impose a severe constraint on the type of documents that are used in practice, but rather provides a subtle distinction between documents that explicitly model cyclic data structures and those that do not.

3.2. Ordered Documents

The class of Ordered documents are defined by Schemas whose components form a partially ordered set (poset) with respect to the direction of the QName references in the Schema components. That is, the component graph of a Schema of an Ordered document is acyclic. This is more formally defined as follows.

Definition 3.2 *Let D_1 and D_2 denote two top-level Schema components (elements, attributes, simpleTypes, complexTypes, etc.). Then, we say that D_1 is lexicographically less than D_2, denoted $D_1 \prec D_2$ if (a ref or QName in) D_2 refers to D_1. Let $\prec\!\!\prec$ denote the transitive relation defined by*

$$\begin{array}{lll} D_1 \prec\!\!\prec D_2 & \textit{if} & D_1 \prec D_2 \\ D_1 \prec\!\!\prec D_3 & \textit{if} & D_1 \prec D_2 \prec\!\!\prec D_3 \end{array}$$

Therefore, the Schemas of an Ordered document contain components D_i that form a poset with respect to the relation $\prec$.

The class of Ordered documents is an important class, because powerful DFA-based parsers can be developed for this class of documents. This will be further discussed in Section 4.

3.3. The XML Document Hierarchy

The proposed classification forms a document subset hierarchy as follows.

Theorem 3.3 *The Type 0, 1, 2, and 3 documents form a subset hierarchy $\mathcal{C}_3 \subset \mathcal{C}_2 \subset \mathcal{C}_1 \subset \mathcal{C}_0$. (Provided that elements with matching ID and REF attributes are defined by matching Schema components.)*

Proof. It is not difficult to see that $\mathcal{C}_2 \subset \mathcal{C}_1$, and $\mathcal{C}_1 \subset \mathcal{C}_0$. To prove the $\mathcal{C}_3 \subset \mathcal{C}_2$ relation, suppose that a cyclic ID-REF reference exist in an Ordered Type 3 document. Let E_1 and E_2 be two elements in the cyclic chain, as defined by Definition 3.1. Let D_1 be the top-level definition containing the definition for E_1 and let D_2 be the top-level Schema component containing the definition for E_2, and let D_a be the top-level Schema component containing the definition for the ancestor element of E_1. Then, $D_2 \prec D_1$, because E_1 refers to an element that is the ancestor of E_2, and $D_1 \prec D_a$, because D_a is the ancestor definition of element E_1. Furthermore, $D_a \not\prec D_2$ or $D_2 = D_a$, because there is an element in the ID-REF chain from element E_2 that has a definition that refers to D_a, the ancestor element that has the ID attribute. Hence, an Ordered document cannot contain ID-REF cycles by proof of contradiction. □

The XML document hierarchy is used to define a category of XML parsers. Generic non-validating XML parsers are able to parse Type 0 documents. Validating XML parsers are able to parse Type 1 documents and reject those that are invalid according to the validation rules.

We will demonstrate that very efficient DFA-based parsers can be generated for the Type 4 Ordered documents. The Schema-specific parsers implement state-based parsing rules derived from the Schema to significantly speed up document processing.

4. Generating DFA-Based Parsers from Ordered Schemas

The `wsdl2DFA` tool automatically generates Flex and DFA descriptions for the DFA-based parser implementation. The Flex description is further processed by Flex to implement a scanner (see Figure 1).

4.1. Constructing the Flex Description

A Flex description consists of three sections:

```
%{
C declarations
%}
Regular expressions
%%
Actions
%%
```

The first section is reserved for the C declarations of the C action statements in the third section. The second section contains the definitions of common regular expressions for possible inclusion in the third section. The third section defines the scanner's actions as C statements that are executed upon recognizing the specific character sequences described by the regular expressions. For more details, see [1, 7].

When `wsdl2DFA` constructs the Flex description for a WSDL or Schema document, the content of the first and second sections is fixed. The code generated for the second section introduces the basic building blocks of an XML document as described below.

Elements and attributes are separated by white space as defined by the following regular expression called `blank`:

```
blank   [ \t\v\n\f\r]*
```

This regular expression can be adapted to match other non-relevant character sequences, such as the UTF8 BOM sequence. The following regular expression defines a token that resembles a NCName or LocalPart of a QName:

```
name    [^>/:= \t\v\n\f\r]+
```

For convenience, the `name` expression represents a non-empty sequence of characters excluding white space and `>`, `/`, `:`, and `=`. Based on the `name` expression, we define a regular expression for optional namespace prefixes:

```
qual    {name}:|""
```

The optional `qual` expression is used for parsing qualified and unqualified names with the following regular expression that defines an opening element tag `<` followed the optional qualifier:

```
open    <{qual}
```

Elements contain optional attributes, so we define a regular expression that sweeps over the element attributes until the end of an opening element tag is found:

```
close    [^>]*>
```

Finally, CDATA is represented with the following regular expression that collects all character data up to, but excluding, a terminating `<`:

```
data    [^<]*
```

The third section in the generated Flex description contains the following predefined scanner actions:

```
{blank}                     // ignore white space
{open}"?"{close}            // ignore declaration
{open}"!"{close}            // ignore comment
{open}"/"{close}            POP
{open}"Header"{close}       PUSH return HEADER;
{open}"Body"{close}         PUSH return BODY;
{open}{name}"/"{close}
{open}{name}{close}         PUSH
{data}                      return DATA;
<<EOF>>                     return EOF;
```

The first three actions ignore the white space that separates the elements, and ignores any XML declarations and comments. The fourth action pops an element's ending tag (i.e. `</X>`). The fifth and sixth action match and push the SOAP Header and Body element beginning tag (the reason for this will be made clear in the next section). The seventh action ignores any elements with empty content (i.e. `<X.../>`). The eight action pushes an element's beginning tag. The ninth action returns element data to the driver, and finally the tenth action returns EOF to the driver.

These actions define the default behavior of the scanner, which is to scan over a document and only return SOAP Header and Body elements, DATA, and EOF to the driver. Anything else is not returned to the driver. The PUSH and POP operations keep track of the node nesting level of the document's elements and may implement other document traversal-related bookkeeping activities such as maintaining a namespace binding stack (see also Section 7).

To add Schema-specific components to the Flex description for parsing and handling of SOAP/XML by the driver, the `wsdl2DFA` tool takes a WSDL or Schema and adds to the Flex description the regular expressions describing all elements found in the WSDL/Schema. These Schema element definitions are collected from all parts of (a set of related) Schemas, including top-level element Schema components and local elements. For example, suppose a Schema contains the following components:

```
<element name="getQuote" type="tns:getQuoteType"/>
<complexType name="getQuoteType">
  <sequence>
    <element name="symbol" type="xsd:float"/>
  </sequence>
</complexType>
```

Then the following two entries are added to the Flex description:

```
{open}"getQuote"{close}     PUSH return ELT_getQuote;
{open}"symbol"{close}       PUSH return ELT_symbol;
```

These actions return the `getQuote` and `symbol` elements to the driver. Any other elements, except the SOAP Header and Body elements, will not be returned to the driver. The reason is that any logic required for handling these extra elements can be performed by the PUSH and POP operations in the scanner.

4.2. Constructing the DFA

The DFA generated from the Schema is the driver program for the scanner. The driver acts as a Schema-optimized XML parser with a state-based approach to restrict parsing to the relevant elements of a document. The DFA forms the parser's state graph with transitions labeled with the token values produced by the scanner, which consist of the scanner's action values (elements, attributes, and data) and a node nesting level indicator. For example, BODY (2) indicates the SOAP Body element at document node nesting level 2. The node nesting level indicates the depth of the elements and data scanned from the document root. The level indicator is required to ensure that invalid documents, i.e. documents that do not match the Schema, can be rejected. The level indicator also guides the DFA's transitions without requiring the DFA to explicitly model transitions on ending element tags, which reduces the DFA to about half the size required otherwise.

The DFA template for a SOAP/XML message is shown in Figure 3. The Schema of the message is translated into

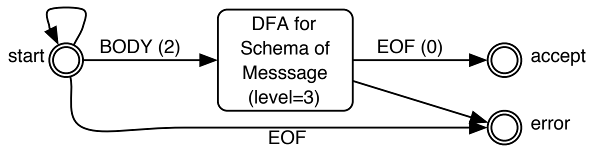


Figure 3. DFA for SOAP-Enveloped Messages

a DFA for parsing XML content at level 3, which is inside the SOAP Envelope and Body. A transition on BODY (2) to the DFA of the Schema takes place when a SOAP Body element is found[1]. After parsing the SOAP Body with the DFA constructed for the message, the transition to the final accepting state is expected to take place on an EOF (0).

The SOAP Header processing is not shown in the DFA in Figure 3. A minimal SOAP Header processor must at least implement the SOAP 1.1/1.2 rules for parsing mandatory Header entries indicated by `mustUnderstand` attribute values. To this end, a Flex description and DFA can be constructed for the SOAP Header components.

The DFA construction proceeds by recursively replacing Schema components with their DFA representations. The DFA representations for commonly used Schema elements and types are shown in Figure 4. Empty transitions (caused by `minOccurs=0`) will be closed. Because the Schema is required to be Ordered, the recursive translation process cannot enter a cycle and will therefore terminate.

The generated DFA-based parser generates events for the application (see Figure 2). These events contain element data, including the XPath expressions pointing to the data.

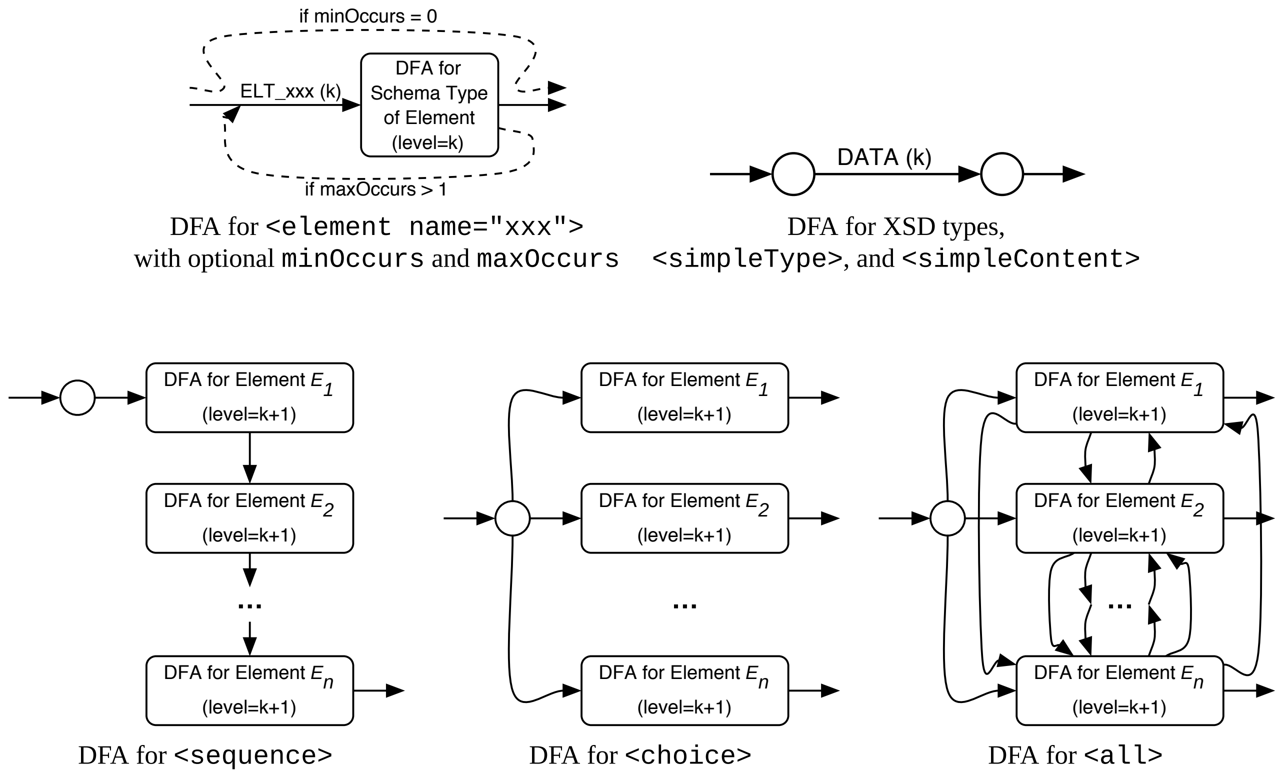
5. Example Applications

This section presents two case studies on two example applications. The first example describes a parser for an echoString service in detail. A performance comparison for echoString message parsing will be presented in Section 6. The second example investigates the suitability of the approach for parsing the WS-Security protocol.

5.1. Case Study 1: The echoString Service

Figure 5 depicts the Schema of the echoString SOAP/XML request message (defined in a WSDL description as a document style message). The `echoString` message element contains a child element `input` of type XSD string. An extensibility element was added to illustrate support for extensibility in our approach.

1 The SOAP Envelope element is ignored. The root Envelope element was omitted to reduce complexity. It can be included in the scanner and DFA if necessary.


if minOccurs = 0
ELT_xxx (k)
DFA for Schema Type of Element (level=k)
if maxOccurs > 1
DATA (k)
DFA for Element E_1 (level=k+1)
DFA for Element E_2 (level=k+1)
...
DFA for Element E_n (level=k+1)

DFA for `<element name="xxx">` with optional `minOccurs` and `maxOccurs`

DFA for XSD types, `<simpleType>`, and `<simpleContent>`

DFA for `<sequence>`

DFA for `<choice>`

DFA for `<all>`

Figure 4. DFAs for Schema Elements and Types

```
<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"/>
        <any namespace="##any" minOccurs="0"
                               maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

Figure 5. The echoString Message Schema

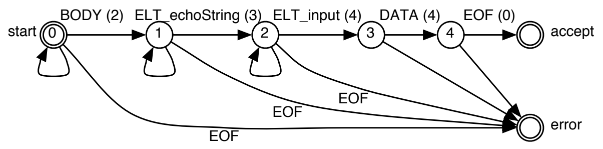
The `wsdl2DFA` tool generates the Flex description shown in Figure 6. For this example, the PUSH and POP operations are simply keeping track of the node nesting level in the document. The level indicator is used for controlling the DFA transitions.

The source code of the DFA generated by `wsdl2DFA` is shown in Figure 7. The `yylex` function returns the next token from the Flex scanner shown in Figure 6. Transitions in the state graph are determined by the token value and the current level indicator.

The `event("echoString/input", yytext)` call returns an XPath expression and string data to the server application. Note that the XPath expression is a pre-compiled constant string in this case. The string data `yytext` is produced by the scanner and contains the CDATA of the `input` element. Because the

```
%{
#include "echoStringDFA.h"
#define PUSH    level++;
#define POP     level--;
%}
blank   [ \t\v\n\f\r]*
name    [^>/:= \t\v\n\f\r]+
qual    {name}:|""
open    <{qual}
close   [^>]*>
data    [^<]*
%%
{blank}                     // ignore white space
{open}"?"{close}            // ignore declaration
{open}"!"{close}            // ignore comment
{open}"/"{close}            POP
{open}"Header"{close}       PUSH return HEADER;
{open}"Body"{close}         PUSH return BODY;
{open}"echoString"{close}   PUSH return ELT_echoString;
{open}"input"{close}        PUSH return ELT_input;
{open}{name}"/"{close}
{open}{name}{close}         PUSH
{data}                      return DATA;
<<EOF>>                     return EOF;
%%
```

Figure 6. Flex Specification for echoString



```
int echoStringDFA()
{ int token, state = 0;
  while ((token = yylex()) != EOF)
    switch (state) {
    case 0: if (token == BODY && level == 2)
              state = 1;
            break;
    case 1: if (token == ELT_echoString && level == 3)
              state = 2;
            break;
    case 2: if (token == ELT_input && level == 4)
              state = 3;
            break;
    case 3: if (token != DATA || level != 4)
              return error("Invalid input value");
            event("echoString/input", yytext);
            state = 4;
            break;
    case 4: if (token == EOF && level == 0)
              return ACCEPT;
            return error("Invalid message");
    }
  return error("End of file");
}
```

Figure 7. DFA for echoString

`echoString` and `input` elements do not carry attributes (as described by the Schema), the parsing of these elements do not result in events. This eliminates the overhead of unnecessarily calls to application functions.

The echoString service implementation of the `event` function extracts the string data. After this extraction, the service returns a response in the form of a SOAP/XML message constructed with libc `sprintf`. The `sprintf` function is much faster than printing the contents of a DOM. However, a DOM provides greater flexibility and gives assurance that the XML output is valid.

5.2. Case Study 2: WS-Security

The WS-Security standard [5] provides transport-level authentication, encryption, and digital signatures. WS-Security is based on the XML Signature standard [14] and the XML Encryption standard [13].

The XML Signature Schema components are acyclic. Therefore documents of the Signature Schema are Ordered. This can also be observed from the informal description of the Signature structure. Expressed in shorthand form, the Signature element of the XML Signature standard has the following structure (where `?` denotes zero or one occurrence; `+` denotes one or more occurrences; `*` denotes zero or more occurrences; and the empty element tag means the element must be empty):

```
<Signature ID?>
    <SignedInfo>
      <CanonicalizationMethod/>
      <SignatureMethod/>
      (<Reference URI? >
        (<Transforms>)?
        <DigestMethod>
        <DigestValue>
      </Reference>)+
    </SignedInfo>
    <SignatureValue>
   (<KeyInfo>)?
   (<Object ID?>)*
 </Signature>
```

The Signature Schema is used by the Encryption Schema, but not vice versa. Documents of the Encryption Schema are Ordered, which can also be observed from the informal description of the EncryptedData structure. Expressed in shorthand form, the EncryptedData element has the following structure:

```
<EncryptedData Id? Type? MimeType? Encoding?>
   <EncryptionMethod/>?
   <ds:KeyInfo>
     <EncryptedKey>?
     <AgreementMethod>?
     <ds:KeyName>?
     <ds:RetrievalMethod>?
     <ds:*>?
   </ds:KeyInfo>?
   <CipherData>
     <CipherValue>?
     <CipherReference URI?>?
   </CipherData>
   <EncryptionProperties>?
 </EncryptedData>
```

The `ds:KeyInfo` element is defined in the Signature Schema and extended with `EncryptedKey` and `AgreementMethod` elements. Because extensions are not explicitly part of the Schemas, the DFA has transitions from the `KeyInfo` element to any other element *that does not create a cycle*. This includes a transition to `EncryptedKey`, but excludes a transition to `EncryptedData`, because this is a parent element of `KeyInfo`. Note that `EncryptedData` elements cannot be children of the `KeyInfo` element anyway, so this implementation is correct.

An example application of the high-performance DFA-based SOAP/XML message parser for WS-Security headers is shown in Figure 8. The figure suggest using a high-

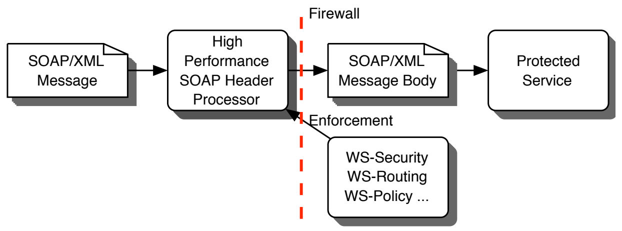


Figure 8. An Example High-Performance SOAP Header Processor

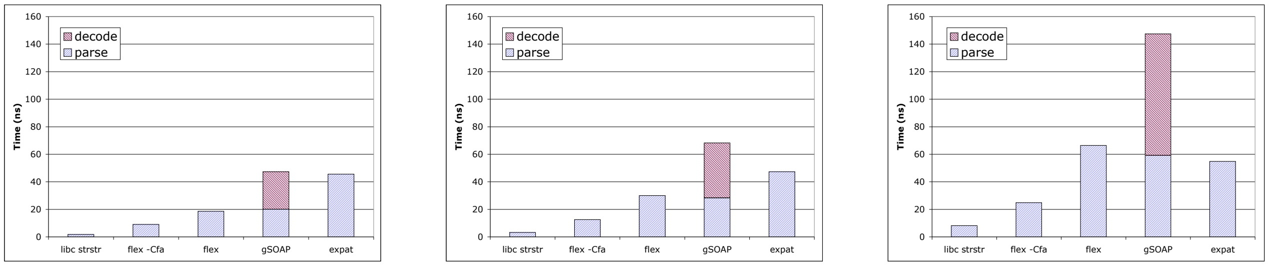


Figure 9. Performance of echoString Parsing and Decoding for $n = 16, 256, 1024$ (800MHz G4)

performance SOAP Header processor to enforce security, routing, and policy decisions. The processor parses SOAP message headers, scanning them for compliance with the desired Web services policies, to verify security tokens and authentication information, perform routing decisions, and so on. This pre-processing must be fast, so the SOAP message body of a compliant message can be passed to a server with a short delay, while malformed messages can be stopped at the firewall boundary without impacting the performance of the service.

6. Performance Results

The performance of the echoString example application of the first case study described in the previous section is compared to the performance of a full-blown SOAP/XML Web services toolkit. The performance was measured on a 800MHz G4 PPC processor, using gcc 3.3 options -O2, Flex 2.5.4. The raw XML parsing performance was measured on memory-resident messages, so network latencies are not taken into account. In addition, the `input` string size n was varied between 16, 256, and 1024.

To set a lower bound on the parsing speed that can be achieved, the libc `strstr` and `strchr` functions were used to extract the contents of the `input` element by simply scanning the XML message for this element. These libc functions should not be considered realistic parsing alternatives, because string searches are fragile and cannot be used to distinguish elements with identical tag names that belong to different parts of a message, such as in a SOAP Header.

Figure 9 shows the performance (elapsed time in nanoseconds) of libc `strstr`, the performance of the DFA-based parser built with a scanner produced with Flex options -Cfa, the performance of the parser with the default configuration for Flex, the performance of an echoString parser built with gSOAP 2.5, and the performance of the eXpat [3] XML parser. The eXpat parser is considered one of the fastest streaming XML parsers. The gSOAP toolkit is an efficient implementation of Web services standards in C and C++ [10, 11]. The compiler-based toolkit effectively translates a Schema to a grammar and generates schema-specific recursive descent parsers to parse XML documents defined by the Schema [9]. Performance studies have shown that the toolkit is efficient [2, 6, 8]. The decoding part of gSOAP shown in the figure indicates the initialization time of the toolkit and the latency for deserializing the message, which involves decoding the input string into dynamically allocated memory.

The Flex -Cfa options are used to generate an optimized scanner that is faster but has a larger code size. The optimized Flex-DFA-based parser is about twice as fast as the gSOAP parser (excluding the decoding phase). The non-optimized DFA-based parser performance is comparable to gSOAP. Note that the DFA-based parser is slower than the ”parser” that uses `strstr` and `strchr`, but the difference is only about a factor of two for larger strings.

We also compared the performance to Apache Xerces for C++ 2.4.0 with all options turned off to improve speed (no validation, no namespaces, no schema support, and no constraint checking). The performance of Xerces is about 290ns for $n = 16$, which is about 30 times slower than the DFA with Flex -Cfa.

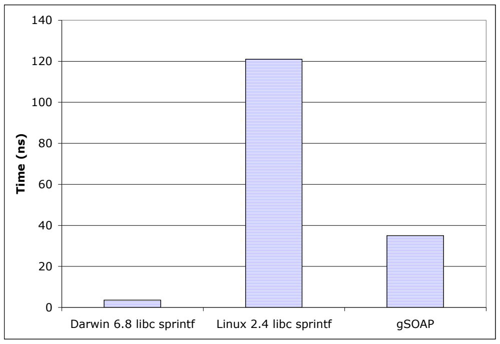


Figure 10. Performance of echoString Response Output for $n = 1024$ (800MHz G4)

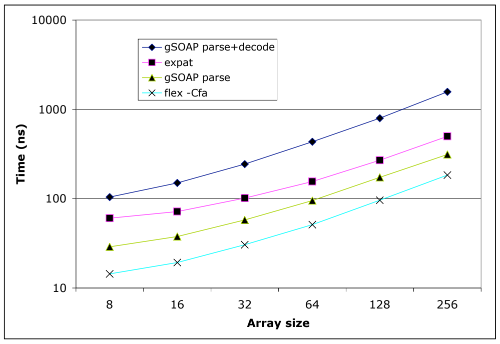


Figure 11. Performance of echoStringArray Request Parsing (800MHz G4)

Figure 10 shows the performance of a SOAP/XML response message generated by a lib `sprintf` call compared to gSOAP. Generating SOAP/XML responses using templates based on `sprintf` is fast with Darwin 6.8, but appears to be slow with Red Hat Linux 2.4 for messages with string length 1024 and higher. Some effort from the application developer is required to ensure that the messages are compliant. Most of the time spent by gSOAP occurs in the initialization phase, which involves the setup and population of a hash table for data analysis to determine co-referenced objects and to detect data graph cycles prior to the XML serialization of the data.

Figure 11 shows the performance of SOAP/XML parsing of echoStringArray messages that contain an array of strings with array sizes ranging from 8 to 256, where the XML representation of the array elements is 16 bytes. The DFA-based parser is about 2 times faster than gSOAP's parser and about 4 to 5 times faster than eXpat.

7. Conclusions

One important issue not addressed in detail in this paper is how to implement XML namespaces [12] support in a DFA-based parser. In short, the XML namespaces standard enables extension mechanisms through namespace qualification of elements and attributes. Consider for example the following XML fragments:

```
<a xmlns="X">
  <b xmlns="Y">...
  <c>...</c>
  <c>...</c>
</a>
```
Does not require NS matching

```
<a xmlns="X">
  <b xmlns="Y">...
  <b>...</b>
</a>
```
Requires NS matching

```
<x:a xmlns:x="X"
     xmlns:y="Y">
  <y:b>...</y:b>
  <x:b>...</x:b>
</x:a>
```
Requires NS matching

The first fragment has an element `a` with two child elements `b` and `c` with distinct names. Therefore, a specialized XML parser that is generated from a Schema can determine the content of the elements by the distinct names of the elements, without requiring namespace matching. The second and third examples contain two child elements `b` in element `a`. Namespace bindings must be used to correctly parse the content of these elements.

To implement XML namespaces, the PUSH and POP operations must be extended with namespace push operations of `xmlns` bindings on a namespace stack, and pop operations for bindings that are out of scope. In addition, the DFA must be extended with namespace information on transitions requiring namespace matching.

The presented parsing technique cannot handle documents of non-Ordered Schemas (those that exhibit cyclic Schema components). However, a quick investigation shows that such documents can in fact be parsed if the level indicator on the DFA transitions is changed from an absolute node nesting depth to a relative node nesting depth with respect to the level of the previously accepted element. With this modification, parts of the DFA can be reused allowing cyclic Schemas to be translated. However, an implementation of this proposed extension has not yet been built to corroborate this claim.

References

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

[2] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *proceedings of the 11th IEEE International Symposium on High-Performance Distributed Computing*, 2002.

[3] J. Clark. eXpat XML parser. http://expat.sourceforge.net.

[4] D. Davis and M. Parashar. Latency performance of SOAP implementations. In *2nd IEEE International Symposium on Cluster Computing and the Grid*, 2002.

[5] IBM and Microsoft. WS-Security specification. Technical report, IBM and Microsoft, 2002. http://msdn.microsoft.com/ws/2001/10/Routing/.

[6] C. Kohlhof and R. Steele. Evaluating SOAP for high-performance business applications: Real-time trading systems. In *proceedings of the 2003 International WWW Conference*, Budapest, Hungary.

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

[8] R. van Engelen. Pushing the SOAP envelope with Web services for scientific computing. In *proceedings of the International Conference on Web Services (ICWS)*, pages 346–352, Las Vegas, 2003.

[9] 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*, Nicosia, Cyprus, 2004.

[10] R. van Engelen and K. Gallivan. The gSOAP toolkit for web services and peer-to-peer computing networks. In *2nd IEEE International Symposium on Cluster Computing and the Grid*, 2002.

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

[12] W3C. Namespaces in XML. http://www.w3.org/TR/REC-xml-names.

[13] W3C. XML Encryption. http://www.w3.org/TR/xmlenc-core/.

[14] W3C. XML Signature. http://www.w3.org/TR/xmldsig-core/.

[15] W3C. XML XPath. http://www.w3.org/TR/xpath.