Published in the proceedings of the International Symposium on Web Services and Applications (ISWS) 2004.

Constructing Finite State Automata for High-Performance XML 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 describes a validating XML parsing method based on deterministic finite state automata (DFA). XML parsing and validation is performed by a schema-specific XML parser that encodes the admissible parsing states as a DFA. This DFA is automatically constructed from the XML schemas of XML messages using a code generator. A two-level DFA architecture is used to increase efficiency and to reduce the generated code size. The lower-level DFA efficiently parses syntactically well-formed XML messages. The higher-level DFA validates the messages and produces application events associated with transitions in the DFA. Two example case studies are presented and performance results are given to demonstrate that the approach supports the implementation of high-performance Web services.

1. Introduction

Several studies on the performance of SOAP [5, 7] reported findings that suggest that SOAP is inefficient for high-end data transport. However, several recent performance studies have shown that the gSOAP toolkit [14] for C/C++ Web services is efficient [3, 7, 12]. The gSOAP toolkit project was the first project to design and implement a compiler-based SOAP/XML engine that adopts an efficient schema-specific XML parsing method [13, 14] that is compliant with the SOAP 1.1/1.2 specification's RPC encoding and DOC/LIT messaging styles.

In contrast to the recursive descent parsing method [1] adopted by gSOAP for the parsing and validation of XML messages, the approach presented in this paper adopts deterministic finite state automata (DFA) to effectively reduce computational requirements for parsing SOAP/XML messages. To improve the efficiency and reduce code size, a two-level DFA architecture for parsing and validation is used. The lower-level DFA efficiently parses syntactically well-formed XML while the higher-level DFA is used for validation and event notification. Validation is accomplished by encoding the XML parser's admissible states as a DFA. This paper describes how the DFAs are automatically constructed by our code generator, which takes a WSDL (Web Service Description Language) document or set of schemas as input and produces the DFA source code to build a Web services application. This paper also presents a performance study demonstrating that the approach supports the implementation of high-performance Web services.

Most closely related to the work presented in this paper is the finite-state parsing technique proposed in [2], which considers merging 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 an essential requirement for SOAP compliance. Furthermore, it is known that the conversion of the non-deterministic automaton to a deterministic automaton with the superset construction algorithm in their approach may result in exponentially growing space requirements [1], possibly leading to scalability problems. In contrast, our two-level DFA approach does not suffer from this problem, because we do not consider merging all parsing aspects into a single complicated DFA structure.

Similar to the DFA-based XML parser architecture presented in this paper, the architecture of our earlier gSOAP toolkit intentionally separates the low-level XML parsing from the higher-level deserialization of data structures to support high-performance and to ensure type safety by constraint validation. The gSOAP compiler-generated parser encodes the restrictions on elements and attributes, such as matching (qualified) element and attribute names and enforcing occurrence constraints. Checking of these constraints proceeds automatically at runtime. Because validation is tightly integrated into the generated parser, validation does not need to be enforced separately, e.g. at the application level which incurs additional overhead [11].

More specifically, the construction of a schema-specific parser with gSOAP is illustrated with an example. Consider the following schema complexType declaration:

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

```
<complexType name="data">
 <sequence>
  <element name="val" type="xsd:int" minOccurs="0"
                                          maxOccurs="10"/>
  <element name="scale" type="xsd:float" minOccurs="0"
                                           maxOccurs="1"/>
 </sequence>
 <attribute name="name" type="xsd:string" use="required"/>
 <attribute name="unit" type="xsd:string" use="optional"/>
</complexType>
```

This `data` complexType contains a sequence of up to ten `val` elements of type XSD int, an optional element `scale` of type XSD float, a required attribute `name` of type XSD string, and an optional attribute `unit` of type XSD string. The gSOAP toolkit produces an annotated C/C++ header file for the `data` complexType:

```
class ns1__data
{ public:
    std::vector<int> *val   0:10;
    float            *scale 0:1;
   @std::string      *name  1;
   @std::string      *unit  0;
};
```

The use of an intermediate header file representation of an XML schema allows familiarization with the mapped data types and constraints by developers who are not experts in XML. The annotated header file defines the `data` class in an XML namespace defined by the `ns1` namespace prefix. The class members include an STL vector of `val` elements with 0:10 indicating the min:max occurrence constraints, and a `scale` element. Attribute members are declared with the @ prefix. The `data` class has an attribute `name` with minimum occurrence 1, and an attribute `unit` with minimum occurrence 0, i.e. the attribute is optional. The gSOAP compiler takes the header file and produces an optimized recursive descent parser for the XML representation of the `data` type, where occurrence constraints are part of the parser's transitions.

The remainder of this paper is organized as follows. Section 2 describes the design and implementation of the schema-specific DFA-based parsing approach. Section 3 presents two examples case studies demonstrating the construction process. Section 4 presents the performance improvement by comparing parsing speeds. The paper is summarized in Section 5 with some concluding remarks.

2. A DFA Parser Generator

This section introduces the DFA-based parser generator. An overview of the system is given and technical details of the implementation are discussed.

2.1. Overview

Figure 1 depicts the DFA generator `wsdl2DFA` developed for this project. 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 parsing and validation. Flex is an automatic generator tool for high-performance scanners [1, 8]. Flex is frequently used by compiler writers to develop scanners that break up a character stream (source code) into a sequence of tokens in the front-end of a compiler. For efficiency reasons Flex scanners are internally implemented as DFAs. 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 XML messages and schemas defined in the WSDL to efficiently parse and decode Web service message invocations for a SOAP/XML Web service application.

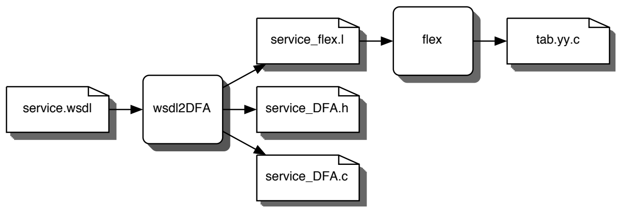


Figure 1. DFA-Based Parser Generator

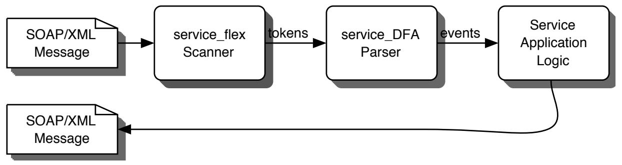


Figure 2. High-Performance Web Service

The Web service application receives a sequence of events from the parser corresponding to the parser's actions. This is somewhat similar to a SAX parser implementation. This event information includes XPath [17] descriptions and data containing the opening XML element tags (with attributes) and the content of elements and attributes.

2.2. Constructing the XML Scanner

The `wsdl2DFA` tool constructs a Flex scanner description based on a WSDL or schema document. This Flex description has three sections (for more details on Flex, see [1, 8]). The content of the first and second section is fixed and independent of the Web service. 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 (REs).

The generated Flex description has the following structure:

```
%{ ... %}
whsp    [ \t\v\n\f\r]
name    [ˆ>/:= \t\v\n\f\r]+
qual    {name}:|""
open    <{qual}
stop    >
skip    [ˆ/>]*
data    [ˆ<]*
xmln    xmlns(:{name}|"")=(\"[ˆ"]*\"|\'[ˆ']*\')
attr    {qual}{name}=(\"[ˆ"]*\"|\'[ˆ']*\')
%x ATTS
%%
{whsp}                     // ignore white space
{open}"?"{skip}{stop}      // ignore declaration
{open}"!"{skip}{stop}      // ignore comment
{open}"/"{skip}{stop}      UP
{open}"Body"               DOWN return BODY; BEGIN(ATTS);
// ... definitions of XML element names and actions
{open}{name}{skip}"/"{stop}
{open}{name}{skip}{stop}   DOWN
{data}                     return DATA;
<ATTS>{whsp}               // ignore white space
<ATTS>{xmln}               PUSH
<ATTS>{attr}               return ATTR;
<ATTS>{stop}               BEGIN(INITIAL);
<*><<EOF>>                 return EOF;
%%
```

Elements and attributes are separated by white space as defined by the `whsp` RE. This RE can be adapted to match other non-relevant character sequences, such as the UTF8 BOM sequence. The `name` RE defines a token that resembles a NCName or LocalPart of a QName. The optional `qual` RE is used for parsing qualified and unqualified names with the `open` RE that defines an opening element tag `<` followed the optional qualifier. The `stop` RE indicates the closing `>`. The `skip` RE sweeps over the element attributes until the end of an opening element tag is found. Character data is represented with the `data` RE that collects all character data up to, but excluding, a terminating `<`. The `xmln` and `attr` REs scan attribute-value pairs, where the `PUSH` operation is used to populate the namespace binding stack with the values of the `xmlns` attributes.

Actions are provided in the section marked `%%`. The fourth action pops an element's ending tag (i.e. `</X>`). The fifth action matches the SOAP Body element beginning tag. The `DOWN` and `UP` operations keep track of the node nesting level of the document's elements. This is followed by the schema-specific element names and actions. The seventh action ignores any elements with empty content (i.e. `<X.../>`). The eight action pushes an arbitrary element's beginning tag. The ninth action returns element data to the driver. The `<ATTS>` actions scan attribute content. The `PUSH` and `UP` operations maintain a namespace binding stack to handle `xmlns` namespace bindings.

The schema-specific REs and actions are added to the Flex description by the `wsdl2DFA` tool. This includes all element names 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:

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

Two actions are added to the Flex specification:

```
{open}"getQuote"  DOWN return ELT_getQuote; BEGIN(ATTS)
{open}"symbol"    DOWN return ELT_symbol; BEGIN(ATTS)
```

The actions return the `getQuote` and `symbol` elements to the driver of the scanner. Any other elements, except the SOAP Body elements, will not be returned. Any logic required for handling these extra elements can simply be performed by the `DOWN` and `UP` operations, when required.

2.3. Ordered Schemas

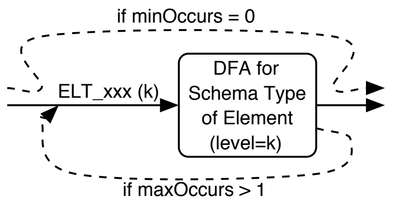
The pure DFA-based parsing technique is suitable for ordered (acyclic) schemas. The components of an acyclic schema form a partially ordered set (poset) with respect to the direction of the QName references in the schema components. That is, the component graph of a schema is acyclic. This is more formally defined as follows.

Definition 2.1 *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. The components D_i of an* ordered schema *form a poset with respect to the relation $\prec$.*

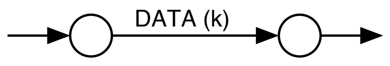
The DFA approach does not permit cyclic schemas in general, i.e. those schemas in which the schema components form a directed graph with at least one cycle. A push-down automaton (PDA) is required to permit the parsing of cyclic schemas. The PDA can lift the power of the parser from regular grammars to context-free grammars. The result of this would be similar to the gSOAP parsers that are based on the recursive descent parsing method.

2.4. Constructing the DFA of the Validating Parser

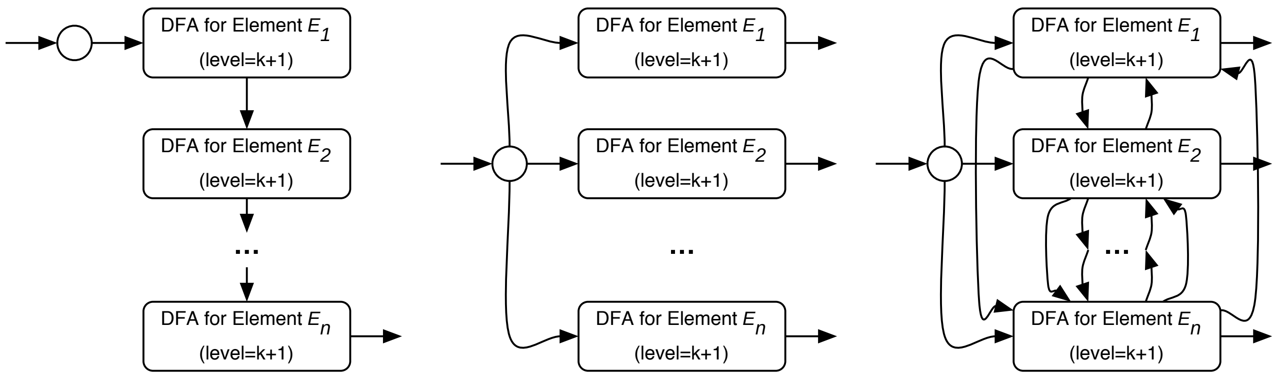
The higher-level DFA generated from a schema (of a WSDL) is the driver that acts as a schema-optimized XML parser. This higher-order DFA implements an efficient state-based approach to XML parsing by considering only the relevant XML elements of a document and to enforce validation constraints. The DFA 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 guides the



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



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


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 `<sequence>` DFA for `<choice>` DFA for `<all>`

Figure 3. DFAs for Schema Elements and Types

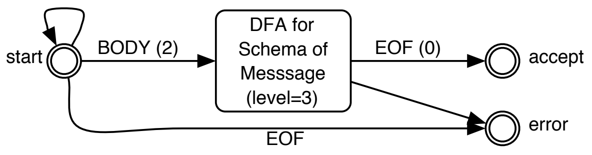


Figure 4. DFA for SOAP-Enveloped Messages

DFA's transitions. The level indicator also enables the automatic handling of ending element tags (which reduces the DFA to about half the size required otherwise). The level indicator is also used to reject invalid documents.

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 3. Empty transitions (caused by `minOccurs=0`) will be closed. Because the schema must be ordered, the recursive translation process cannot enter a cycle and must therefore terminate.

The DFA template for a SOAP/XML message is shown in Figure 4. The schema of the message is translated into 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).

3. Example Applications

This section presents two case studies. The first example describes a parser for an echoString service in detail. A performance comparison for echoString message parsing will be presented in Section 4. The second example investigates the approach for parsing the WS-Security protocol.

3.1. 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.

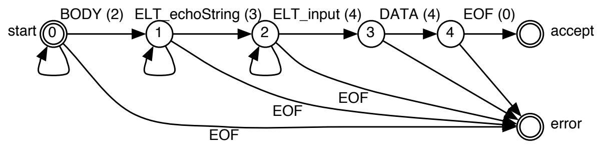
1 The SOAP Envelope and optional Header elements are ignored. The Envelope was omitted to reduce complexity. A minimal SOAP Header processor must implement rules for parsing mandatory Header entries indicated by `mustUnderstand'` attributes. The Envelope and Header elements can be included in the parser if necessary.

```
<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 source code of the DFA generated by `wsdl2DFA` is shown in Figure 6. The `yylex` function returns the next token from the Flex scanner. Transitions in the state graph are determined by the token value and the 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 `echoString` and `input` elements do not carry attributes, the parsing of these elements do not result in events. This eliminates the overhead of unnecessarily calls to application functions.



```
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 6. DFA for echoString

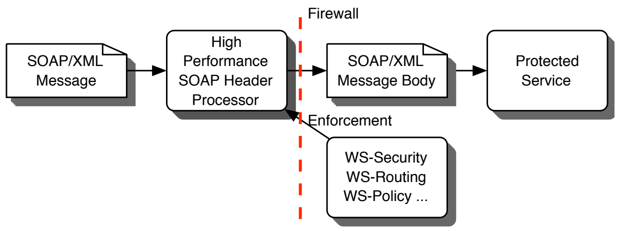


Figure 7. High-Performance Header Parser

The echoString service implementation of the `event` function extracts the string data. After this extraction, the service returns a response in the form of an XML message constructed with libc `sprintf`. The `sprintf` function is much faster compared to populating and emitting a DOM.

3.2. Case Study 2: WS-Security

The need for a high-performance SOAP Header parser is shown in Figure 7. The figure illustrates the use of a SOAP Header processor to enforce security, routing, and policy decisions. Invalid messages can be stopped at the firewall boundary. This preprocessor must be fast to allow the SOAP message body of a compliant message to be passed to a server without impacting the performance.

The WS-Security standard [6] provides transport-level authentication, encryption, and digital signatures. WS-Security is based on the XML Signature standard [16] and the XML Encryption standard [15].

The Signature schema components are acyclic, i.e. the Signature schema is ordered, which can also be observed from the informal short-hand description of the Signature structure (where `?` denotes zero or one occurrence; `+` denotes one or more occurrences; `*` denotes zero or more occurrences; and the empty tag means the element is 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. The Encryption schema is ordered, which can be observed from the informal short-hand description of the EncryptedData structure shown below:

```
<EncryptedData Id? Type? MimeType? Encoding?>
   <EncryptionMethod/>?
   <ds:KeyInfo>
```

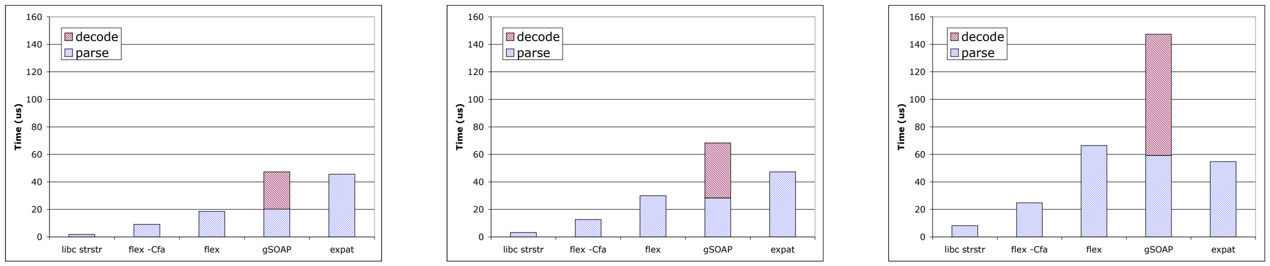


Figure 8. Performance of echoString Parsing and Decoding for $n = 16, 256, 1024$ (800MHz G4)

```
    <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.

4. 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 the full-blown gSOAP 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. Therefore, network bandwidth and latencies are not taken into account.

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 elements occurring in a SOAP Header.

Figure 8 shows the performance (elapsed time in μs) 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 [4] XML parser. The `input` string size n was varied between 16, 256, and 1024.

The eXpat parser is considered one of the fastest streaming XML parsers. The decoding part of gSOAP shown in the figure indicates the initialization time of the toolkit and the latency for deserializing the message and instantiating the C data structures, 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.

The performance of the 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) was also compared. The performance of Xerces is about 290ns for $n = 16$, which is about 30 times slower than the DFA with Flex -Cfa.

Figure 9 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.

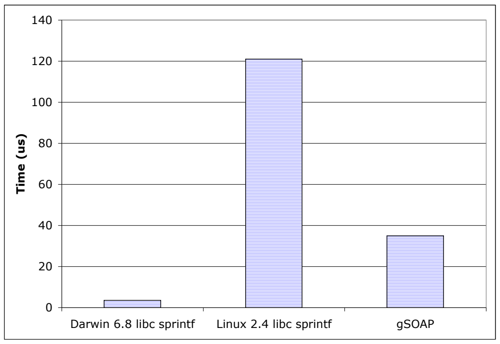


Figure 9. Performance of echoString Response Output for $n = 1024$ (800MHz G4)

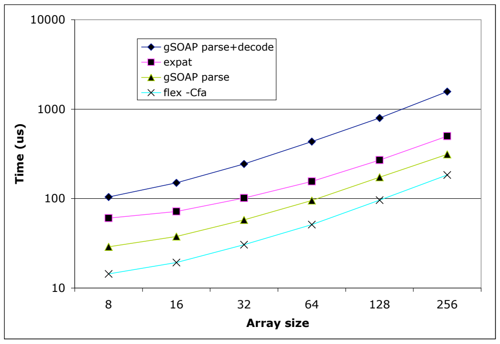


Figure 10. Performance of echoStringArray Request Parsing (800MHz G4)

Figure 10 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 (without data decoding) and about 4 to 5 times faster than eXpat.

5. Conclusions

This paper introduced an XML parsing approach based on a DFA for scanning XML input and a DFA for parsing and validation. The two-level DFA approach can be used in high-performance environments. Performance results show an increase of about a factor 2 to 5 speed increase over other fast implementations.

Another choice for representing XML structures and structural constraints are tree grammars [9, 10]. Tree grammars provide an effective means for XML analysis and manipulation, but are ill suited for low-level parsing. Therefore, tree grammars were not considered in this paper.

References

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

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

[3] 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.

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

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

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

[7] 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.

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

[12] 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.

[13] 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.

[14] 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*, 2002.

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

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

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