A Compiler-Based Approach to Schema-Specific Parsers for XML
Indiana University Tech Report No. 592

Kenneth Chiu and Wei Lu

Indiana University
{chiuk,welu}@cs.indiana.edu

Abstract

The Extensible Markup Language (XML) has become the de facto standard for interoperable data representation. Its human-readable, general syntax provides wide applicability and ease-of-use. These same characteristics, however, complicate the efficient processing of XML, and have created concerns about the performance of XML for distributed systems such as Web services. XML parsers are generally considered either validating or non-validating. Validating parsers compare the input document to a template, also known as a schema, for XML instances. This provides a kind of type-safety to distributed systems. Currently, validation is considered expensive, but we posit that schema information can actually be used to speed-up parsing. This paper develops a framework for such parsing, called schema-specific parsing, and presents preliminary results that show this is indeed a viable approach to high-performance XML parsing.

1 Introduction

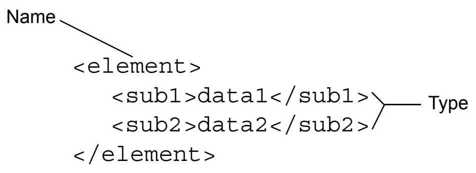
The Extensible Markup Language (XML) is a text-based, human-readable format for structured information. Aided by the ubiquity of HTML, a related format, it has become the de facto standard for interoperable data representation.

In addition to its use for the storage of data, XML is also the natural choice as a transfer syntax for Web services-based middleware, further increasing its prevalence. XML has also found applications in the scientific computing community in standards such as the Earth Science Markup Language [3], a trend that is being accelerated by the ongoing convergence of Grid computing and Web services, as seen in standards such as the WS-Resource Framework [4].

The features of XML that foster interoperability, however, also hinder parsing efficiency. This has spurred concerns about its performance, especially for middleware. In this paper, we introduce a compiler-based approach to XML parsing, called *schema-specific parsing* (SSP) [1], that we believe will significantly reduce the computational burden of parsing XML.

2 XML Schema

XML organizes data into a tree-like structure. The primary units is the element, which is a named node in the tree. Each element can have a set of name-value pairs known as attributes. Within each element is data known as content. Element content can be in the form of child elements, or text.



```
<element>
  <sub1>data1</sub1>
  <sub2>data2</sub2>
</element>
```

Figure 1. The element name is not part of its type. The type defines the contents of the element, and any attributes.

As evident from this description, XML documents are very general, and can be arbitrarily structured. Many real programs, however, can only sensibly accept of a small subset of possible XML documents. Thus, some way to describe the set of acceptable XML documents greatly facilitates the development and maintenance of XML-based systems. These descriptions are generically known as schemas. An XML document that belongs in the set described by the schema is known as an instance of the schema.

An XML schema is simply a pattern, or template, for XML documents. There are number of standards for XML schema, including DTD, and RELAX NG [2], but the most popular is XML Schema[1] [8], which we use in this paper.

An XML Schema is itself an XML document. A full description of XML Schema is beyond the scope of this paper, but we briefly mention a few concepts that are used later.

2.1 Types

Each element declared in a Schema has a name, and a type. The name declares the name of the element, while the type defines a “pattern” for the contents of the element. Note that the type of an element does not include its name, as shown in Figure 1. This is similar to the idea in the C programming language that a `struct` definition defines the contents of the `struct`, but a variable of that `struct` type can have any name.

[1]In this document, we will use “Schema” to refer to schema according to the XML Schema specification, and “schema” to refer to XML schema in general.

2.2 Recursive Schema

A schema can contain types that are self-referential. The corresponding instance may be nested to any given depth. For example, the following Schema fragment defines a recursive type named `recursive`, and one element named `nested` of type `recursive`.

```
<complexType name="recursive">
   <choice>
      <element name="base" type="string">
      <element name="nested" type="recursive">
   </choice>
</complexType>
<element name="nested">
```

The `<choice>` element specifies that valid content is either of the two contained elements. The `<base>` element is the base case of the recursion, and terminates the nesting. The `<nested>` element is recursive. The following is a valid instance of the schema.

```
<nested>
   <nested>
      <base>A string</base>
   </nested>
</nested>
```

2.3 Occurrence Constraints

A type may specify that an element has constraints on how many times it can occur. These are known as occurrence constraints. Occurrence constraints require that the number of times an element appears in an instance be maintained in a count. For example, the following Schema fragment indicates that the `<item>` element may appear between 2 to 5 times within the `MyType` type.

```
<complexType name="MyType">
   <sequence>
      <element name="item" type="string"
       minOccurs="2" maxOccurs="5"/>
   </sequence>
</complexType>
```

2.4 Namespaces

To prevent name collisions, XML supports the notion of namespaces. These are similar to namespaces in programming languages. A schema can specify that the elements belong in a given namespace. Elements with the same name but in different namespaces do not conflict.

Namespaces are specified using a namespace prefix, which must be defined in either a parent element, or the element where it is used.

```
<parent xmlns:ns="http://ns1.org/">
   <ns:child xmlns:ns="http://ns2.org/"/>
</parent>
```

In the fragment, the namespace of the `<child>` element is actually `http://ns2.org` rather than `http://ns1.org`.

3 XML Parsing

The processing of XML in an application can be divided into three stages.

1. **Well-formedness**. The first stage is syntactic, and addresses whether or not the document is well-formed XML. This stage is specifically recognized by the XML specification.
2. **Validity**. The second stage addresses whether or not the structure is a valid instance of a given schema. This stage is also specifically recognized by the XML specification.
3. **Application**. In the third stage, the application actually uses the data in the XML.

These three stages are conceptual, and can be implemented in various ways. In one scenario, a general XML parser parses the XML into some kind of data structure representation of the XML. A validation pass is then made over the XML. The XML data structures are then presented to the application. Typically, the first two stages are performed by a validating XML parser.

4 Schema-Specific Parsing

A common perception within the Web services community is that XML parsing is slow, and that XML validation is even slower. Thus, the complete three-stage division described above is often not directly implemented. Instead, the validation is performed implicitly by the application. A general XML parser passes unvalidated XML to the application, which then will implicitly checks its validity (conformance to the schema) by simply trying to use it. Essentially, the second and third stages have been merged into one, as shown in Figure 2.

A priori, however, we find no reason to believe that validating parsers are inherently slower, and we posit that this belief is in fact a myth. XML schemas contain information that may actually speed-up the lexical analysis and parsing of XML documents, if exploited correctly. For example, a well-known technique for increasing performance is to exploit the frequency distribution of the input by optimizing the most common cases. An XML processor can utilize the

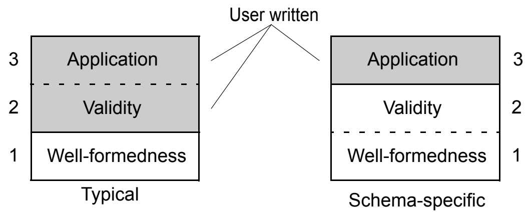


Figure 2. Typical applications merge the second and third layer. Schema-specific parsers merge the first and second layers. This merging presents greater opportunities for optimizations. Merging layers 1 and 2 is normally not practical, because the resulting code would be complex, difficult to maintain, and require modification every time the schema changed.

schema to infer the input frequency distribution for this purpose. Without this, an XML parser must be prepared to handle any kind of input. This is further explained in the Section 6, which shows a code fragment of our parser.

We thus believe that instead of merging the second and third stages, as is commonly done, we should instead merge the first and second stages. The merged parser is what we call a schema-specific parser.

Compared to merging the second and third stages, this also improves interoperability and simplifies debugging by essentially providing a form of type-safety in the middleware layer. Incorrect XML documents are caught automatically, rather than passed on for the application to detect. This reduces the amount of tedious and error-prone validation code that must be present in any robust application.

4.1 Schema Compilation

Merging the first and second stages reduces abstraction and encapsulation costs, but the resulting code is complex, and therefore difficult to develop and maintain. By generating the code through a compilation process, however, the difficulty can be ameliorated.

Schema compilers treat the schema as source code, and compile it into a target language. Some previous work [10] has then interpreted the target language (which is an intermediate, non-native form) with a XML schema machine of some kind. Other work has maintained the parsing as a two-stage process: the first is a schema-independent pass to verify well-formedness, and the second is a schema-dependent validation pass. This separation results in additional computational costs [9][7].

The basic idea behind schema-specific parsing is to generate actual machine code that simultaneously parses and validates the XML. This code is then executed directly, rather than interpreted, which can be much faster. The situation is analogous to the speed-up that can be obtained by hardware execution of native machine code, rather than virtual machine interpretation of some kind of intermediate byte code.

The resulting schema-specific parser only accepts XML documents which are valid instances of the source schema. All other XML documents are rejected.

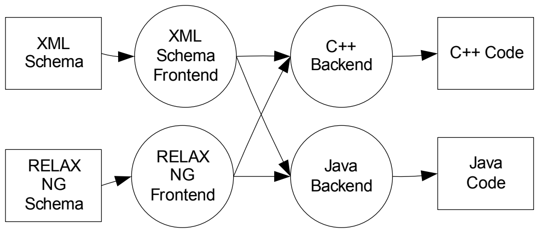


Figure 3. Our architecture divides the compilation process into a front-end and back-end, with an intermediate representation used in between. This allows different front-ends to work with different back-ends.

Rather than directly generating the parser code from the schema, we adopt an approach similar to that of traditional compilers. A front-end first parses the schema into an intermediate representation, and a back-end then generates code from this intermediate form. This simplifies the design, and produces opportunities for optimizing transformations to be performed on the intermediate form.

This also supports the development of different back-ends for different target languages and purposes. For example, one back-end could generate Java byte code, while another could generate C++. Furthermore, even within the same target language, different back-ends can generate different kinds of code. For example, one C language back-end might generate a parser optimized for speed, while another might generate code optimized for power-efficiency on a mobile device. Even with the same target language, an optimization on one hardware architecture may be a pessimization on a different one, suggesting that different code be generated for each architecture.

Similarly to how different compiler front-ends can generate the same intermediate language from different source languages (like the architecture for gcc), we can also develop front-ends for different schema languages. Our current work focuses on the XML Schema language.

4.2 Generalized Automata

Many choices are available for the intermediate representation of the schema. The issues are similar to choosing an intermediate language for a compiler. One that is too high-level might not expose enough low-level details to support various kinds of useful manipulations and transformations. On the other hand, one that is too low-level hides other kinds of opportunities.

One choice for an intermediate representation is finite automata (FA). Finite automata would allow some kinds of optimizations to be performed. However, FAs do not have sufficient power to validate many aspects of XML schemas, such as occurrence constraints. Thus, various kinds of ad hoc extensions would be required. These extensions would hinder reasoning about the intermediate representation, because our formal model (the FAs) no longer closely model our actual intermediate representation. Transformations and

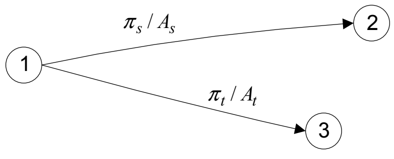


Figure 4. A generalized automata (GA) has a predicate π and list of actions A with every transition. If the predicate is true, the transition is enabled and may be taken. If taken, the actions must be executed. Each predicate has a readset which indicates which variables it depends on. Each action has a writeset, which specifies which variables it changes.

algorithms valid for FAs would not be valid for our intermediate representation.

Furthermore, the ad hoc extensions would likely be different for different schema languages. This would make it difficult to implement different front-ends for different schema languages without requiring changes to the back-ends.

Another choice for an intermediate representation is high-level constructs like tree grammars [5][6]. Tree grammars, though, do not adequately represent the low-level aspects of parsing, such as lexical analysis. Thus, they hinder the ability to merge well-formedness checking, which is highly lexical, with validation.

We thus believe a more powerful intermediate representation simplifies the overall architecture, both in concept and and implementation, and choose as our intermediate representation a generalization of pushdown automata (PDA) we call generalized automata (GA)[1]. Rather than just a stack, a GA has a finite set of variables. Each transition has a predicate over the variable set, rather than just an input symbol and a stack symbol. Each transition also has an ordered list of actions. A transition can be taken if the predicate is true for the current variable values, and when taken, the actions are executed.

GAs encompass FAs. An FA is a GA with one variable, the input buffer. A FA transition labelled with an input symbol a is equivalent to a GA transition with one predicate which returns true if the current input symbol equals a. The GA version of an FA transition has one action which consumes the next input symbol. An epsilon transition corresponds to a GA transition with a predicate that is always true, and no actions.

Similarly, PDAs can be mapped to GAs. A PDA is a GA with two variables, the input buffer and the stack.

Because GAs are powerful enough model all schema constraints without ad hoc extensions, they can serve as a formal model to reason about the schema compilation process, which simplifies the development of algorithms.

Formally, a GA is defined by the nonuple

$$M = (Q, U, \delta, \Pi, A, q_0, \xi_0, F, E) \tag{1}$$

[1]We are aware that the term *generalized automata* is already being used for something else, but we have yet to think of a better term.

where Q is a set of states, U is a set of variables, δ is a transition function, Π is a set of predicates over U, A is a set of actions over U, q_0 is the start state, ξ_0 is the initial configuration, F is a set of final states, and E is a set of final configurations.

A configuration represents the values of all the variables, and is an element from the set $V_1 \times \ldots \times V_n$, where V_i is the set of values that can be stored in variable u_i.

4.2.1 Transition Function

The transition function maps from the state and a configuration to a finite set of pairs. Each pair is a new state, and a list of actions. The mapping is to a set, rather than a single pair to accommodate nondeterminism.

$$\delta(q, \xi) \to \{(q_1, A_1), (q_2, A_2), \ldots\} \tag{2}$$

Here, q is the current state. A_1 and A_2 are lists (ordered sets) of actions.

Since the number of configurations is possibly infinite, the transition function is not defined by enumeration. Rather, we use an equivalent graph-based formulation similar to that used for FA. Each edge in the graph is represented by a quadruple (q, p, π, A), where q is the source vertex, p is the target vertex, π is a predicate, and A is a list of actions, as shown in Figure 4. We say that the transition is *enabled* when π is true for the current configuration. We also say that a state is enabled when the source vertex is clear from context. The GA may take any enabled transition nondeterministically. Upon taking a transition, it must execute the list of actions (in order) associated with that transition.

When the context is clear, we will also refer to the predicate of a state as shorthand for the predicate of the transition to that state.

Each action writes to a set of variables, and each predicate reads from a set of variables. The function **writeset**(A) returns the set of variables written to by the list of actions A. The function **readset** when applied to a predicate returns the set of variables read by the predicate. The functions **readset** and **writeset** may be applied to transitions and has the expected meaning. The function **readset**(q) returns the union of the readset of the predicates of all transitions going out from q.

Given a transition t, **source**(t) returns the source vertex, and **target**(t) returns the target vertex. Also, **action**(t) and **pred**(t) return the actions and predicate, respectively.

The function **trans**(q) returns the transitions of the state q.

4.2.2 Instantaneous Description

Note that in the GA model, the state of the computation, in the English language sense, is not just the state of the machine as defined above. The state of the computation includes the current configuration. The same is also true for FAs, in the sense that the state of the computation also depends on the contents of the input buffer at that instant, and not just the formal state.

Thus, the instantaneous description represents the complete state of the machine at any one moment. It is a snapshot of the running machine.

$$(q, \xi) \tag{3}$$

where q is the current state, ξ is the current configuration. Upon taking a transition, the machine updates the instantaneous description by executing the actions.

4.3 Schema Predicates and Actions

The GA model does not define the actual predicates and actions, but rather only that they have readsets and writesets, respectively. For SSP, we chose predicates and actions appropriate for XML Schema, some of which are listed in Table 1. These predicates are recognized by the back-end code generators which can then generate the correct code to implement the semantics of the Schema instance.

	Name	Purpose
Predicates	match *a*	True if the next input symbol is *a*.
	call_site *s*	True if the call site was s. This is used to match the return transition to the return address.
	occurrence	Used for controlling the matching under occurrence constraints.
Actions	consume	Consume a symbol from the input buffer.
	call	Push a context on the stack.
	return	Pop a context.
	attr_start	Beginning of an attribute.
	attr_char	An attribute character.
	attr_end	End of attribute.
	value_start	Beginning of the attribute value.
	value_char	An attribute value character.
	value_end	End of attribute value.
	pref_start	Beginning of namespace prefix.
	pref_char	Namespace prefix character.
	pref_end	End of namespace prefix.

Table 1. Predicates and actions used for generating parsers for XML Schema.

Types are handled specially by the front-end. The front-end generates one group of states for each Schema type *T*. When an element is defined of type *T*, a **call** action to the start of *T* is created, and a **return** action from the end of *T* back to the element's end tag is also inserted. The return transition has a predicate that is only enabled when call site corresponds to the return state (Figure 5).

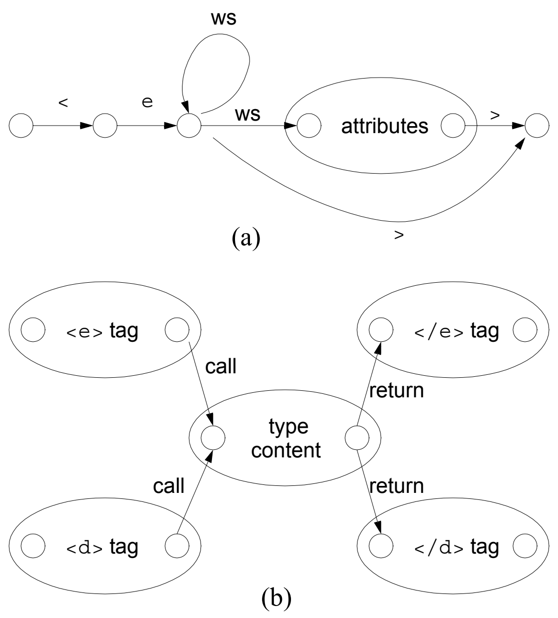


Figure 5. The GA fragment for a start tag is shown in (a). The notation **ws** represents a set of transitions for the various whitespace characters. The **attributes** machine represents a set of states for parsing the attributes. In (b), we see how a single type content machine serves multiple elements. The **call** transition modify the configuration in such a way that the correct **return** transition is enabled. The exact details of this modification are determined by each back-end.

This supports recursive Schemas, and also eliminates the combinatorial state explosion caused by nesting of types.

Note that the valid attributes of an element are part of the type, but are parsed in the start tag machine, not the shared content type machine. This is because the namespace is not known until the end of the start tag is seen, so the correct content type is not know until the end of the start tag.

We have defined our predicates and actions for the XML Schema, but we believe that other schema languages can be accommodated with modest changes.

4.4 Nondeterministic GA (NGA) to Deterministic GA (DGA) Conversion

The GA generated by the front-end is non-deterministic. This simplifies the front-end, and any transformations such as might be performed by optimization algorithms. Code generated from a non-deterministic GA would have to simulate the non-determinism, however, which is inefficient. Thus, we convert the NGA to a DGA before code generation. Of course, not all NGAs are convertible to DGAs. In practice, however, we have not found this to be a problem, and, since we control the front-end, we can tweak the output of the front-end if it does ever pose an issue.

The algorithm is based on the subset construction algorithm used to convert NFAs to DFAs. The first pass is analogous to epsilon-closure, and is called move-compression. The second pass constructs the actual subsets. The implementation queries the backend for the actual read- and write-

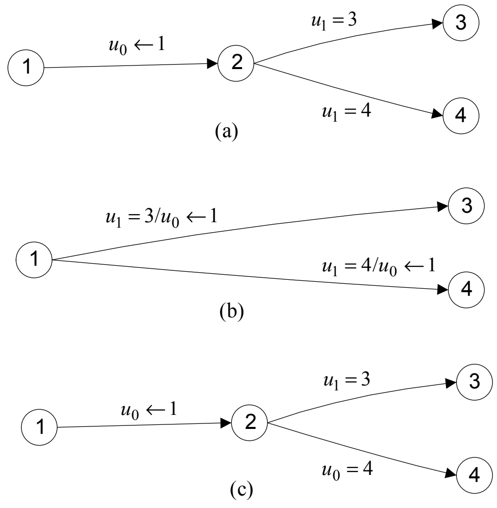


Figure 6. The writeset of the 1-2 transition in (a) does not intersect with the readset of state 2, so the transitions can be compressed, as shown in (b). The resulting transitions are guaranteed to result in an equivalent machine. In (c), the transitions cannot be compressed, because the transition from 2-4 depends on the value of u_0. Note that even the path 1-2-3 cannot be compressed, even though the predicate of 2-3 does not depend on u_0, because resulting machine would not be equivalent.

sets. This allows a single implementation of various GA algorithms to work with multiple backends.

4.4.1 Move-Compression

The basic idea in subset construction for NFAs is that the constructed DFA has states that represent multiple NFA states. This allows the DFA to simultaneously follow multiple paths in the NFA. Paths are abandoned when subsequent input discriminates those paths as dead ends.

With FA, this construction is relatively straightforward, because there is only one action, which is to consume an input symbol. With generalized automata, however, two transitions enabled by a configuration may have different actions, and the different actions change the configuration in different ways. This means that two transitions cannot be merged if the actions are different, since we cannot simultaneously maintain multiple configurations within the GA model.

To reduce the number of such conflicts, we first make a move-compression pass. This pass compresses transition paths, so that two transitions that previously had different actions may now have the same action.

The main idea behind move-compression is that a sequence of transitions can be combined into one transition if the actions of the first transition do not interfere with the predicate of the second. After move-compression, the invariant is that **writeset**(t) must intersect with **readset**(**target**(t)).

```
Let todo be a stack of states representing work to do.
Push the start state on to todo.
While there is a state p left in todo:
    Pop todo.
    Mark p.
    For each transition t out of p:
        If the writeset(t) does not intersect with
        readset(target(t)), then:
            For each transition s out of target(t):
                Insert a new transition from p to target(s) with
                predicate of pred(t)^pred(s) and actions
                action(t) concatenated with action(s).
            Remove t.
    If t was removed, then push p back on to todo; else push all
    unmarked targets of p on to todo.
```

Also note that a transition path with a loop in the first transition should be skipped.

4.4.2 Subset Construction

After the move compression pass, we next perform subset construction similarly to the NFA subset construction algorithm. In this algorithm, all transitions that are enabled for the same input symbol are grouped together into a subset. Because the input alphabet is relatively small, these subsets can be easily determined by enumeration or sorting. For GAs, however, the configuration space is much larger, and is in fact infinite.

We therefore generalize the NFA subsets by defining an equi-enabled set of a state p to be a set of transitions out of p such that there is at least one configuration, where all transitions in the set are enabled and all out of the set are disabled. We define the superset of a state p to be the set of all equi-enabled sets. (A more detailed treatment can be found in the appendix.) The subset construction algorithm for GAs is then

```
Let dstates be a set of states to containing the states of the
newly created DGA.
Insert the start state into dstates.
While there is an unmarked state p in dstates:
    Mark p.
    For each equi-enabled set S in the superset of p:
        Lookup state q representing S in dstates; if not found,
        create a new state unmarked state q for S in dstates.
        Add a transition from p to q with predicate and
        actions the same as the transitions in S.
```

This algorithm requires that all outgoing transitions in the same subset have the same list of actions. So far, we have not found that to be a limitation, but we will describe a more sophisticated algorithm in a later publication to handle such cases.

Generating the supersets is difficult in general. We currently generate these in a case-by-case fashion, but we out-

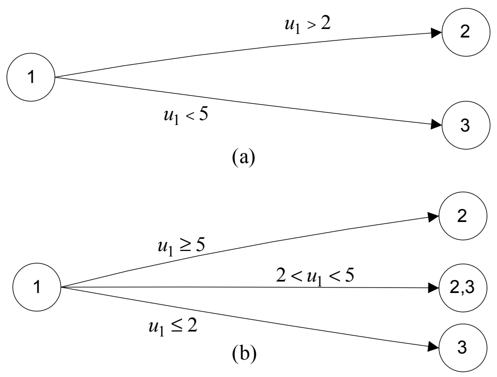


Figure 7. If as shown in (a), u_1 is in the range (2, 5), then both state 2 and state 3 are enabled when in state 1. The machine must thus be in both states until further computation can discriminate the two paths. We thus merge states 2 and 3, as shown in (b). Note that the predicates on transitions 1-2 and 1-3 must also be augmented.

line how we might generate supersets in more general cases in the appendix.

5 Code Generation

The GA may undergo a number of optimizations. For example, isomorphic sub-graphs may be identified. These sub-graphs could be replaced with a single set of states, thus reducing the code size. If necessary, additional Schema predicates and actions may be defined to assisted these optimizations.

The back-end traverses the GA and generates code. Our prototype back-end is a C++ generator designed for speed. We map groups of states which parse types to C++ functions, which will allow us to use the program stack for many of the more expensive actions, thus improving performance and handling recursive schema in a natural, efficient way. For example, counters for occurrence constraints can simply be automatic variables.

6 Preliminary Results

Work is still in progress, but we are able to obtain some initial feedback on our approach. The schema we used is given below (slightly edited for clarity).

```
<schema targetNamespace="http://www.foo.org">
   <element name="elem" type="Type"/>
   <complexType name="Type">
      <choice>
         <element name="sub1" type="string">
         <element name="sub2" type="string">
      </choice>
   </complexType>
</schema>
```

This schema describes an element named `elem` with two possible subelements. One choice is a `<sub1>` subelement and the other is a `<sub2>` subelement. Both possible subelements contain a string. The commented sample below illustrates the generated code. (The whitespace transitions on newline and tab have been deleted for clarity.)

```
// Get next character from buffer.
if ((c = *p++) == 0) { fill(); p = buf; c = *p;}
// This reads the last char of the elem name.
// The 'except' label is an error state.
if (c != 'm') goto except;
if ((c = *p++) == 0) { fill(); p = buf; c = *p;}
// If it's an angle bracket, it is end of tag.
if (c == '>') goto label34;
// If it's a space char, may be attributes.
if (c == ' ' || c = '\r') goto label10;
goto except;
label10:
if ((c = *p++) == 0) { fill(); p = buf; c = *p;}
// Loop on white space.
if (c == ' ' || c == '\r') goto label10;
// Beginning of "xmlns".
if (c == 'x') goto label11;
// Beginning of "attr".
if (c == 'a') goto label16;
// End of start tag.
if (c == '>') goto label34;
goto except;
```

We can see above one way in which schema-specific parsing facilitates exploitation of the input distribution. Since we know that most input will be correct, we can simply insert the expected characters directly into the conditionals. Without schema information, the element name would have to stored to memory for subsequent access, thus increasing the memory bandwidth requirements.

The Schema instance is:

```
<ns:elem xmlns:ns="http://www.foo.org"
 attr="value">
   <sub1>sub1content</sub1>
</ns:elem>");
```

On a 1.7GHz Pentium 4 running Linux, compiled with gcc 3.2 and the `-O` option, our generated parser validates each XML document in.87 microseconds. The documents were streamed from a file, so the time includes file I/O, but does not include any significant overhead for tasks such as opening files or initiating network connections. The parsed XML is not further processed in any way.

We compared this to expat 1.2 compiled with the same options on the same machine. The expat parser required 4.9 microseconds per instance. Thus, our preliminary results show that we are about 5 times faster than expat.

This is encouraging, especially considering that expat is considered to be one of the fastest XML parsers, and that we are comparing a our validating parser against a non-validating parser.

Our initial generated code does not support multiple namespace uses or definitions, and default namespaces. We do not anticipate that supporting these will severely impact our performance, because schema-specific parsing provides the parser with useful information, and because code generation allows techniques that may be too tedious and error-prone for hand-written code.

7 Conclusion

This paper contributes a framework for a compiler-based approach to schema-specific parsing of XML. The framework applies compiler techniques to the parsing of XML, thus improving performance. A simple formal machine, the generalized automata provides a flexible model for developing algorithms and implementations. The generated parser is executed natively by the hardware, rather than interpreted, as in some other validation approaches. Preliminary results are encouraging, and suggest that the approach has promise.

Further work will expand the supported subset of XML Schema, and investigate other kinds of back-ends and front-ends. For large schemas, techniques may need to be provided to improve code locality to avoid instruction cache thrashing. Another issue to address is the user API. That is, we still need to define an efficient interface to provide the user with the actual content of the parsed XML.

8 Acknowledgements

We thank Steve Gabriel for suggesting the superset construction algorithm described in the appendix.

9 References

[1] Kenneth Chiu, Madhusudhan Govindaraju, and Randall Bramley. Investigating the Limits of SOAP Performance for Scientific Computing. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing* (HPDC '02), July 2002.

[2] James Clark and Murata Makoto. *RELAX NG Specification*. December, 2001. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[3] Earth Science Markup Language. http://esml.itsc.uah.edu/.

[4] Globus Alliance, The WS-Resource Framework. http://www-fp.globus.org/wsrf/default.asp.

[5] M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages,* Montreal, Canada, Aug. 2001.

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

[7] Florian Reuter and Norbert Luttenberger. Cardinality Constraint Automata: A Core Technology for Efficient XML Schema-aware Parsers. http://www.swarms.de/publications/cca.pdf.

[8] Henry S. Thompson, et al. *XML Schema Part 1: Structures*. http://www.w3.org/TR/xmlschema-1/.

[9] Henry S. Thompson and Richard Tobin. Using Finite State Automata to Implement W3C XML Schema Content Model Validation and Restriction Checking. In *Proceedings of XML Europe 2003*. http://www.idealliance.org/papers/dx_xmle03/papers/02-02-05/02-02-05.html.

[10] Ning Wang, Peter S. Housel, Guogen Zhang and Michael Franz. An Efficient XML Schema Typing System. Technical Report 03-25. School of Information and Computer Science, University of California, Irvine. Nov., 18th, 2003.

Appendix: Superset Construction

Given a state p, we first partition the configuration space into a set of equivalence classes C_p^i based on the relation R_p, where $\xi_0 \; R_p \; \xi_1$ iff for all t in **trans**(p), $\pi_t(\xi_0) = \pi_t(\xi_1)$

Each C_p^i induces a set of transitions T_p^i, which is the set of transitions enabled by the configurations in C_p^i.

$$\forall \xi \in C_p^i [\forall t \in T_p^i(\pi_t(\xi)) \wedge \forall s \notin T_p^i(\neg \pi_s(\xi))] \qquad (4)$$

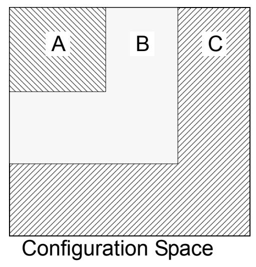
We call each T_p^i an equi-enabled set, and the set of all equi-enabled sets is the superset. An example is shown in Figure 8.

We now outline an algorithm to construct the superset of a state. First, we stipulate that all variables are integer-valued, and that all predicates are boolean expressions comprised of relational operators of the form u **op** n, where n is an integer and u is a variable. Each relational expression then defines a hyperplane which partitions the configuration space into two or three subspaces, depending on whether the relation is an ordering relation or equal relation, respectively.

This suggests that we parse the predicates, and use each relational expression to cut the configuration space along the hyperplane defined by the expression. The cuts are cumulative, so that when finished we have cut the configuration space into rectangular regions such that for every region, we can be assured that the same set of predicates is true.

We then test one configuration from each region on each predicate. The set of true predicates for that one configuration defines an equi-enabled set.

The restriction that the relational expression be of the form u **op** n has not proven to be a problem, but we plan to



Equivalence Class	Enabled Transitions
A	T1, T2
B	T2, T3
C	T1, T2, T3

Figure 8. There are three equivalence at this particular state. Within each equivalence class, the same set of predicates is enabled. This partitioning thus induces three equi-enabled sets: {T1, T2}, {T2, T3}, and {T1, T2, T3}.

address it in future work. The primary complication is that if the expression is of the form *u* **op** *v*, then the cuts are not orthogonal to a dimension, complicating the implementation.