A Compiler-Based Approach to Schema-Specific XML Parsing

Kenneth Chiu and Wei Lu

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

Abstract

The validation of XML instances against a schema is usually performed separately from the parsing of the more basic syntactic aspects of XML. We posit, however, that schema information can be used during parsing to improve performance, using what we call schema-specific parsing. This paper develops a framework for schema-specific parsing centered on an intermediate representation we call generalized automata, which abstracts the computational steps necessary to validate against a schema. The generalized automata can then be used to generate optimized code which might be onerous to write manually. We present results that suggest this is a viable approach to high-performance XML parsing.

1 Introduction

XML is a text-based, human-readable format for structured information. Spurred by the ubiquity of its cousin HTML, it has become the de facto standard for interoperable data representation. XML is also the natural choice as a transfer syntax for Web services-based middleware, further increasing its prevalence.

Some of the features responsible for XML's popularity, however, also affect parsing. This has led to concerns about its performance, especially for middleware. One promising technique is creating parsers specific to a schema [2][5][7], which we call schema-specific parsing.

This paper contributes a compiler-based approach for schema-specific parsing that we believe provides a flexible framework for implementing high-performance, schema-specific parsers. Our framework is centered around an intermediate representation called the generalized automaton (GA). Our intermediate representation serves a purpose similar to that of an intermediate language in a programming language compiler.

First we briefly review XML Schema in Section 2. Then in Section 3, we describe our approach in detail, including schema-specific parsing and generalized automata. Section 4 outlines the processing that occurs for code generation. Performance results are presented in Section 5. In Section 6 we review related work. We conclude in Section 7 with a conclusion and future work.

2 XML Schema

XML documents are general, and can be arbitrarily structured. Many production 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 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 [3], but we have initially chosen to focus on XML Schema[1] [12]. 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 declaration in a Schema associates a name to a type. The name declares the name of the element, while the type specifies the contents of the element. Thus, the name is distinct from the type, which allows a type to be referenced in multiple element declarations.

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

[1] In this document, we will use "Schema" to refer to schema as defined by the XML Schema specification, and "schema" to refer generically to any XML schema.

<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 can constrain the number of times a contained element can occur. These are known as occurrence constraints. 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>
```

3 Schema-Specific 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.
2. **Validity**. The second stage addresses whether or not the structure is a valid instance of a given schema.
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. If the stages are fully articulated, 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. Often, the first two stages are packaged together into what is known as a validating parser.

A common perception among practitioners is that XML parsing is slow, and that XML validation is even slower. Thus, some applications do not implement the complete three-stage division. Instead, a general XML parser passes unvalidated XML to the application, which then implicitly validates it by error checking and possibly exception handling. Essentially, the second and third stages have been merged into one, as shown in Figure 1.

We posit, however, that XML schemas contain information that may actually speed-up the lexical analysis and parsing of XML documents, if exploited correctly. 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.

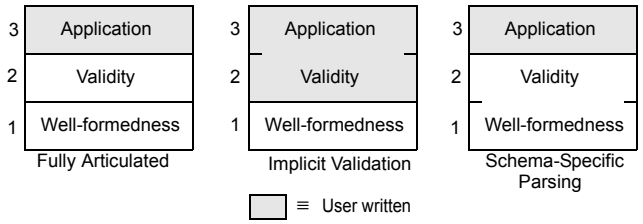


Figure 1. In a fully articulated implementation, each stage is distinct. Some applications forgo a validating parser, and instead implicitly validate the XML during use, thereby essentially merging layers 2 and 3. Schema-specific parsers, on the other hand, merge layers 1 and 2. This merging of layers 1 and 2 is normally not practical, because the resulting code would be complex, difficult to maintain, especially when the schema changes. (The gray stages are user written.)

For example, without schema information element names must be buffered by the lexical analyzer for use by the application or during validation. If schema information is available, element names can be directly resolved to an application-provided, element-specific handler during lexical analysis.

3.1 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. Since the first and second stages require no application-specific processing, however, we can generate the code by schema compilation.

Schema compilers treat the schema as source code, and compile it into a target language. The output may then be interpreted by a validation engine, or executed externally.[1] The resulting parser only accepts XML documents which are valid instances of the source schema. All other XML documents are rejected.

In our approach we output executable code that simultaneously parses and validates the XML. The simultaneous parsing and validation merges the lexical, syntactic, and schema aspects of XML processing into a single code layer, which tends to improve locality and CPU register utilization. This code is then executed directly, rather than interpreted by a processing engine or execution kernel. Analogous to the expected speed-up of compiled object code over interpreted byte code, we also expect a directly executed schema to be faster than an interpreted schema. In [7], Löwe, Noga, and Gaul provide further evidence for this performance gain.

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, analogous to an intermediate lan-

[1] A target language such as Java may in fact be interpreted, but such interpretation would be external and hidden to the XML processing.

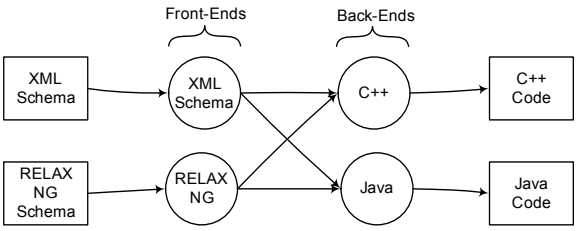


Figure 2. Our architecture divides the compilation process into a front-end and back-end, with an intermediate representation used in between. This allows different front-ends to work with different back-ends. Round shapes represent processes, while rectangular shapes represent data.

guage in programming language compilers, 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, as shown in Figure 2. Our current work focuses on the XML Schema language.

Our approach may be too onerous in situations where the schema change frequently. An interpreted approach as in [14] may be more appropriate in such scenarios.

3.2 Generalized Automata

Many choices are available for the intermediate representation of the schema. The issues are similar to that of 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 discards structural information that might be useful for other kinds of optimizations.

One choice for an intermediate representation is finite automata (FA) [13][14]. Though suitable when used as a higher-level abstraction for schema validation, they are less useful for code generation. Finite automata would allow some kinds of optimizations to be performed. However, FAs do not have sufficient power to validate some aspects of XML schemas, such as occurrence constraints. Thus, various

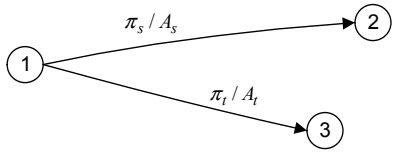


Figure 3. A generalized automata (GA) has a predicate π and list of actions A with every transition. If the predicate is true, the transition is enabled and may be taken. If taken, the actions must be executed. Each predicate has a readset which indicates on which variables it depends. Each action has a writeset, which specifies which variables it modifies.

kinds of ad hoc extensions would be required. These extensions would complicate transformations on the intermediate representation, because the FAs would no longer closely model the generated code. Transformations valid for the FAs would not necessarily result in correct code. This would reduce the benefits of a formal model.

Another choice for an intermediate representation is high-level constructs like tree grammars [4][8][9]. 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, which is primarily structural. Tree grammars also may have some difficulty scaling to arbitrary occurrence constraints, and handling aspects such as namespaces. Tree grammars are powerful models for reasoning about schemas, but perhaps less appropriate for code generation.

Yet another choice is context-free grammars [5][7]. We believe the issues with context-free grammars are similar to those for tree grammars.

We thus believe a different intermediate representation simplifies the overall architecture, both in concept and implementation, and choose as our intermediate representation a generalization of pushdown automata (PDAs) we call generalized automata (GA)[1]. Rather than just a stack, a GA has a finite set of variables. Each variable can hold arbitrary values (without bound). 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. A fragment of a GA is shown in Figure 3.

Each predicate reads from a set of zero or more variables, which is termed the readset. Each action writes to a set of zero or more variables, which is termed the writeset. Actions also have readsets, though we currently assume than an action's readset is equivalent to its writeset. Actions may be compared for equivalence; predicates may also be compared for equivalence.

GAs encompass FAs. An FA is a GA with one variable, the input buffer. A FA transition labelled with an input sym-

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

bol 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. The stack can be treated as a single variable, because we do not place any restrictions on the contents of a variable. The transition function for PDAs can be represented by an appropriate predicate.

Note that GAs do not provide a complete computational model. Any arbitrary computation may occur in the actions, and variables are not restricted. This is a deliberate omission, because our goal here is not a representation for reasoning about the schema itself, but rather a representation for reasoning about the computational steps required to validate a schema. We desire a model *about* computation, not a model *for* computation; we want to manipulate computation, not actually compute.

Thus, we only model the aspects of computation that we believe affect code generation. So we permit actions, predicates, and variables to be defined arbitrarily and external to the model, but include in the model the equivalence relations and ordering dependencies that we believe are significant to the generation of efficient code. Each back-end provides a specific set of actions, predicates, and variables that together define an abstract processor. This processor functions as a CPU specialized for parsing and validating XML.

Seen thusly, GAs extend FAs with information that constrains valid transforms, so that transforms valid on the GA will still result in valid code. Similarly, tree grammars have no mechanism for any reasoning about computation that takes place in extensions that are outside of the tree grammar model. GAs essentially abstract arbitrary computation as actions, while maintaining information useful for reasoning about code generation.

Formally, a GA is defined by the eight-tuple

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

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, and F is a set of final states.

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. The set of all configurations is known as the configuration space. Actions can be viewed as mappings from one configuration to another.

Name	Applied To	Definition
actions	transition t	action list of t
pred	transition t	predicate of t
	transition set T	$\bigcup_{t \in T}$ **pred**(t)
readset	action a	set of variables read by a
	action list A	$\bigcup_{a \in A}$ **readset**(a)
	predicate π	set of variables read by π
	state q	**readset**(**trans**(q))
	transition t	**readset**(**pred**(t))
	transition set T	$\bigcup_{t \in T}$ **readset**(t)
source	transition t	source vertex (state) of t
	transition set T	$\bigcup_{t \in T}$ **source**(t)
target	transition t	target vertex (state) of t
	transition set T	$\bigcup_{t \in T}$ **target**(t)
trans	state q	$\{t:(q =$ **source**$(t))\}$
	state set Q	$\bigcup_{q \in Q}$ **trans**(q)
writeset	action a	set of variables written by a
	action list A	$\bigcup_{a \in A}$ **writeset**(a)
	transition t	**writeset**(**action**(t))

Table 1. Notation used to access various parts of a GA.

3.2.1 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.

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

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 FAs. Each edge in the graph is represented by a four-tuple (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 3. We call such an edge a transition, and say that it is *enabled* when π is true for the current configuration. Note that GA transitions are slightly different from transitions in FAs and PDAs. A single GA transition might be enabled for many different input symbols, which is not the case for FAs and PDAs.

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.

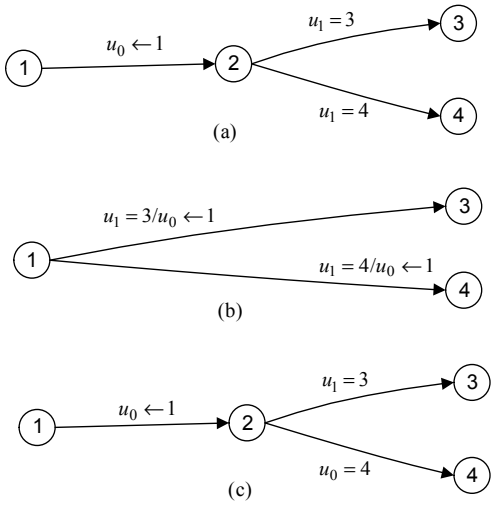


Figure 4. The writeset of the 1-2 transition in (a) does not intersect with the readset of state 2, so the transitions can be compressed, as shown in (b). The resulting transitions are guaranteed to result in an equivalent machine. In (c), the transitions cannot be compressed, because the transition 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.

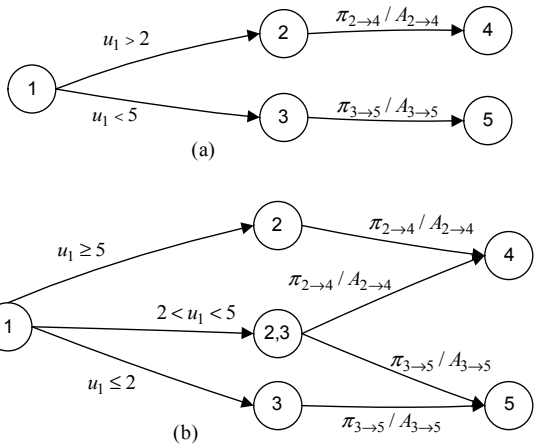


Figure 5. As shown in (a), If u_1 is in the range (2, 5), then both state 2 and state 3 are enabled when in state 1. The machine must thus be in both states until further computation can discriminate the two paths. We thus merge states 2 and 3, as shown in (b). Note that the predicates on transitions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

When the source vertex is clear from context, we say that a state *q* is enabled when the transition from the source vertex to *q* is enabled. 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.

Table 1 explains the notation used to access various parts of GAs.

3.2.2 Instantaneous Description

Note that in the GA model, the state of the computation, in the English language sense, is not just the formal 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 formal state, ξ is the current configuration. Upon taking a transition, the machine updates the instantaneous description by executing the actions.

3.3 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; since we know that not all non-deterministic PDAs can be converted to deterministic PDAs, and PDAs can be represented as GAs. 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 ever does 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 back-end for the actual read- and writesets. This allows a single implementation of various GA algorithms to work with multiple backends.

3.3.1 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 FAs, 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 purpose of this is to remove transitions that are redundant, and to move predicates as early as possible in an execution sequence.

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*)). Move compression is shown in Figure 4.

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.

Our current move compression algorithm does not perform all valid compressions. So far we have not found that to be a problem, but will incorporate further compressions as necessary.

3.3.2 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 not in 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 **target**(*S*) in *dstates*; if not found, create a new unmarked state *q* for target(*S*) in *dstates*. For the new state *q*, $\mathbf{trans}(q) = \mathbf{trans}(\mathbf{target}(S))$.

Add a transition *t* from *p* to *q*. Let e_1 be the AND of all predicates in **pred**(*S*), and e_2 be the OR of all predicates in **pred**(*u*), where $u \in \mathbf{trans}(p) - S$. Then set **pred**(*t*) to $e_1 \wedge \neg e_2$.

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 develop more sophisticated algorithms if necessary. For example, a list of actions can be split between two transitions if there is no readset interference.

Generating the supersets is difficult in general. We currently generate these in a case-by-case fashion, but we outline how we might generate supersets in more general cases in the Appendix.

3.4 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 our initial focus, we chose predicates and actions appropriate for XML Schema, some of which are listed in Table 2. These predicates are recognized by the

	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 2. Predicates and actions used for generating parsers for XML Schema.

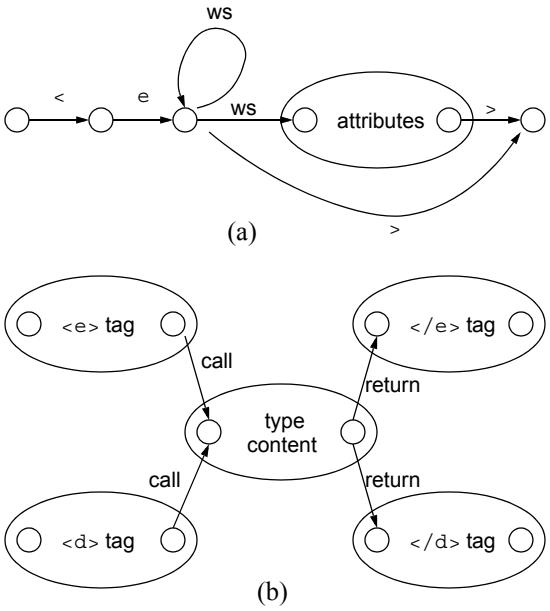


Figure 6. The GA fragment for a start tag is shown in (a). The notation **ws** represents a set of transitions for the various whitespace characters. The **attributes** machine represents a set of states for parsing the attributes. In (b), we see how a single type content machine serves multiple elements. Each **call** transition modifies 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. Note that the type content machine can be mapped naturally to a function in the target language. In this case the **return** transition would simply be the usual **return** statement in the programming language.

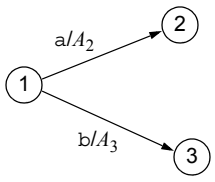
back-end code generators which can then generate the correct code to implement the semantics of the Schema instance. Taken together, these predicates and actions essentially serve as the instruction set of an abstract Schema validation processor.

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 6).

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. The goal is that one set of predicates and actions can be used with a variety of front-ends. A single back-end can then generate parsers for a number of schema languages.




```
LABEL_FOR_STATE1:
    if (c == 'a') {
        // Do actions A2.
        goto LABEL_FOR_STATE2;
    if (c == 'b') {
        // Do actions A3.
        goto LABEL_FOR_STATE3;
    }
    goto ERROR1;
```

Figure 7. The GA fragment on the left is translated to the code on the right. State is maintained implicitly with the program counter (location in code) rather than an explicit state variable. We also encourage the compiler to use immediate operands in the instruction stream.

4 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. We note that after NGA to DGA conversion, the machine resembles a predictive parser, but with transitions that can depend on arbitrary predicates, rather than just the next input symbol.

The back-end traverses the GA and generates code. For each predicate and action, it outputs code it deems necessary for validation. Each back-end will translate the predicates and actions differently. 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 could simply be automatic variables.

We also do not use an explicit state variable to encode the GA state. Rather, we use the program counter to maintain state, and encourage the compiler to include data in the instruction stream as immediate operands, as shown in Figure 7. These techniques tend to improve memory access behavior. Note that a table-driven approach would prevent the 'a' character from being included as an immediate operand.

5 Performance Results

Our prototype implementation currently only accepts only a small subset of XML Schema: string attributes, choice, sequence, complex types. We also accept a single maximum occurrence constraint within a sequence. We do not currently accept namespaces.

We compared our implementation to Libxml2 2.6.7, expat 1.2, and gSOAP 2.5. Due to the highly varied nature of these parsers, these tests are not meant to show that any

parser is necessarily faster than any other, but rather to provide feedback on the potential of our approach.

Libxml was configured with no extra options. The validation was performed through the Reader interface. The main loop simply called `xmlTextReaderRead()` repeatedly after enabling validation. The expat parser was compiled without namespace and DTD support. The program did not examine any of the parsed information. gSOAP was configured with no special options. All code was compiled with the Sun Workshop C and C++ compilers version 5.3 and the -O option.

The Schema we used is given below (slightly edited for clarity).

```
<schema>
   <complexType name="elemType">
      <choice>
         <element name="sub1" type="string"/>
         <element name="sub2" type="string"/>
      </choice>
   </complexType>
   <complexType name="topType">
      <sequence>
         <element name="elem" type="elemType"
          maxOccurs="N"/>
      </sequence>
      <attribute name="attr" type="string"/>
   </complexType>
</schema>
```

This schema describes sequence of `<elem>` elements, of *N* maximum occurrences. Each `<elem>` element has one attribute and two possible subelements. One choice is a `<sub1>` subelement and the other is a `<sub2>` subelement. Both possible subelements contain a string. Since Libxml2 does not accept XML Schema, the following DTD was used:

```
<!DOCTYPE top [
   <!ELEMENT top (elem+)>
   <!ELEMENT elem (sub1|sub2)>
   <!ELEMENT sub1 (#PCDATA)>
   <!ELEMENT sub2 (#PCDATA)>
   <!ATTLIST elem attr CDATA "default">
]>
```

For gSOAP we used `wsdl2h` to generate an C++ header file from the Schema and then generated the stubs and skeletons from the header file with `soapcpp2`.

The code generator is a straightforward translation of the GA. Each vertex is represented by a block of code. Each transition consists of one **if** statement, followed by the actions within the body of the **if** statement. If the vertex only has one transition, we reverse the sense of the **if** statement so that the valid case simply falls through. The occurrence constraint is validated via a local integer counter variable. The commented sample below illustrates the generated code.

```
// If an e, it is the beginning of "elem".
if (c == 'e') {
   consume_input();
   goto label34;
}
// If slash, it is the end of elemType type.
if (c == '/') {
   consume_input();
   return;
}
// If whitespace character, it is beginning of
// optional whitespace.
if (c == ' ' || c == '\r' || c == '\n'
    || c == '\t') {
   consume_input();
   goto label29;
}
goto error3;
```

We can see above one way in which schema-specific parsing facilitates exploitation of the schema. Since we know the tag names, we simply insert the expected characters directly into the conditionals. Without schema information, the element name would have to be stored to memory for subsequent access, thus increasing memory accesses.

We have not settled on a complete user API, but used a SAX-like API with handlers specific to the elements. That is, each element has a different handler, so the application will not need to reexamine the tag name to know which element has been encountered. In this case, calls to non-leaf handlers had no arguments, while the call to the leaf handlers were passed the contents of the element as C strings.

The documents were read from a file located on a tmpfs filesystem, so the tests include system calls for I/O. Time was measured with the `gethrtime()` call. We attempted to avoid including various initialization in the measured time, but cannot guarantee that there was no hidden initialization that was inadvertently included. The parsed XML was not further processed in any way.

The tests were run on a Sun Fire 280R Model 21200 with two UltraSPARC-III+ 1200 MHz CPUs and 4 GB of main memory, and the results are shown in Figure 8 as a log-log graph.

The results show that our approach can be significantly faster than other parsers, both validating and non-validating. Libxml is of course a comprehensive, production-quality parser, so includes much more functionality. We also note that the Libxml Reader API is easier to use than the SAX-like API of our implementation, and likely trades some performance for ease-of-use. gSOAP also incurs the overhead of C++ strings, which can be slow but are more convenient and robust than simply passing a C string to the application. The performance difference between expat and SSP is at least partially due to the additional writes that expat uses to

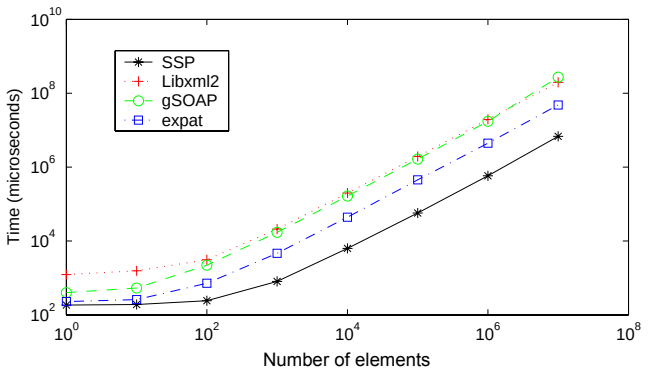


Figure 8. We show here the time to parse a document of a given size in log scale on both axes. The x-axis shows the number of elements in the document. The y-axis is total time for the document in microseconds. Note that the times for 1 element likely includes a costly `read()` system call, which tends to obscure the differences between parsers. For 100,000 elements and over, the difference between expat and SSP was about 7 times. The test was run on a UltraSPARC III+ 1200 MHz CPU.

store the tag name for access by the application during the callback.

Despite these differences that prevent a direct comparision, we nonetheless believe that we have a flexible, viable approach to building high-performance XML parsers. Future work will explore the limits of our approach.

6 Related Work

A number of previous researchers have investigated schema-specific parsing and related techniques.

In our previous work [2], we conducted a preliminary investigation into using SSP for SOAP arrays, and found that performance could in fact be improved.

Thompson and Tobin [13] augmented finite state automata to validate Schema content models. The resulting FSAs, however, may be very large for some occurrence constraints.

Wang et al. [14] compile the schema into a compact form known as Annotated Automata Encoding. This is then interpreted by a kind of FSA. The advantage of this method is that they do not need to recompile any source code when the schema changes. Performance may not be as good as with direct execution, however.

Lee, Mani, and Murata [6] investigate the use of regular tree grammars to analyze schemas and validate instances. They focus on the structure of the elements.

In [11], Reuter and Luttenberger develop cardinality constraint automata, which handle occurrence constraints and `<all>` more efficiently than tree automata. Lexical analysis is still performed separately, however.

In [7], Löwe, Noga, and Gaul investigated the generation of deterministic context-free grammars for DTDs and restricted Schemas. With their aXMLerate toolkit, they found the resulting performance to be quite good, and confirm that schema-specific parsers can actually be faster than non-validating parsers.

Noga, Schott, and Löwe [10] provide a classification of parsers based on control-flow (push vs. pull) and type information access (compiled vs. interpreted). They also present an architecture for lazy (pulled and interpreted) XML processing.

Engelen and Gallivan also use context-free methods [5]. They generate predictive parsers directly as opposed to through an intermediate representation.

7 Conclusion and Future Work

This paper contributes a compiler-based approach to schema-specific parsing of XML. Analogues to compiler design are drawn, where appropriate, and applied to schema compilation. A simple formal machine, the generalized automata, provides a flexible model for developing code optimization algorithms and code generation. The generated parser is executed natively by the hardware, rather than interpreted, as in some other validation approaches. Results suggest that our approach to SSP is significantly faster than non-schema-specific parsers.

Further work will expand the supported subset of XML Schema, and investigate other kinds of back-ends and front-ends. Similarly to compilers, multiple intermediate representations may be useful. For example, it may be appropriate to first use a tree grammar representation to perform some transformations at the schema level. The tree grammar might then be converted to a GA for further manipulation. We also wish to explore the relationship of GAs to structures such as control flow graphs.

Another issue is other schema languages. Other schema languages may have ambiguity issues that will present problems for the NGA to DGA conversion algorithm.

For large schemas, techniques may be needed to improve code locality to avoid instruction cache thrashing. We may wish to find similar sub-graphs, and merge these into one graph. Additional variables may be used to handle small differences between the original sub-graphs.

Another way to reduce code size is through the use of common action elimination. Often all transitions into a state will execute the same action. Rather than duplicate the action, we can simply execute the common action upon entry to the state.

Further design and development of an appropriate user API is also required. The interface should balance efficiency against usability.

8 Acknowledgements

We thank Steve Gabriel for suggesting the superset construction algorithm described in the appendix. We also thank Sriram Krishnan, Aleksander Slominski, and Matt Sottile for useful comments and discussion.

9 References

[1] Boris Chidlovskii. Using Regular Tree Automata as XML Schemas. In *Proceedings of IEEE Advances in Digital Libraries 2000 (ADL 2000)*. May 22 - 24, 2000, Washington, D.C.

[2] 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.

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

[4] Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. http://www.grappa.univ-lille3.fr/tata/.

[5] Robert A. van Engelen and Kyle 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 (CCGrid2002)*. May, 2002. Berlin, Germany.

[6] D. Lee, M. Mani, and M. Murata. "Reasoning about XML Schema Languages using Formal Language Theory". Technical report, IBM Almaden Research Center, RJ# 10197, Log# 95071, Nov. 2000.

[7] Welf Löwe, Markus L. Noga, Thilo Gaul. Foundations of Fast Communication via XML. *Annals of Software Engineering*. Volume 13(1-4), p. 357-379, January 2002.

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

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

[10] Markus L. Noga, Steffen Schott, and Welf Löwe. Lazy XML processing. In *Proceedings of the 2002 ACM Symposium on Document Engineering*. ACM Press.

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

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

[13] 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.

[14] 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. Given two configurations ξ_0 and ξ_1, they are related via $\xi_0 \; R_p \; \xi_1$ 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.

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

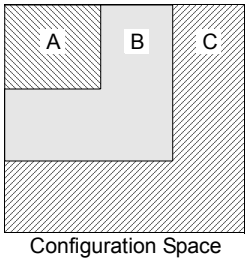
We call each T_p^i an equi-enabled set, and the set of all equi-enabled sets is the superset. An example is shown in Figure 9.

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 address it in future work. The primary complication is that if



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

Figure 9. 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}.

the expression is of the form *u* **op** *v*, then the cuts are not orthogonal to a dimension, complicating the implementation.