Generation of efficient parsers through direct compilation of XML Schema grammars

E. Perkins
M. Matsa
M. G. Kostoulas
A. Heifets
N. Mendelsohn

With the widespread adoption of SOAP and Web services, XML-based processing, and parsing of XML documents in particular, is becoming a performance-critical aspect of business computing. In such scenarios, XML is often constrained by an XML Schema grammar, which can be used during parsing to improve performance. Although traditional grammar-based parser generation techniques could be applied to the XML Schema grammar, the expressiveness of XML Schema does not lend itself well to the generic intermediate representations associated with these approaches. In this paper we present a method for generating efficient parsers by using the schema component model itself as the representation of the grammar. We show that the model supports the full expressive power of the XML Schema, and we present results demonstrating significant performance improvements over existing parsers.

INTRODUCTION

XML has begun to work its way into the business computing infrastructure and underlying protocols such as the Simple Object Access Protocol (SOAP) and Web services. In the performance-critical setting of business computing, however, the flexibility of XML becomes a liability due to the potentially significant performance penalty.

XML processing is conceptually a multitiered task, an attribute it inherits from the multiple layers of specifications that govern its use: XML,[1,2] XML Namespaces,[3] XML Information Set (Infoset),[4] and XML Schema.[5] Traditional XML processor implementations reflect these specification layers directly. Bytes, read off the "wire" or from disk, are converted to some known form (often UTF-16 characters) and tokenized (UTF stands for Universal Text Format). Attribute values and end-of-line sequences are normalized. Namespace declarations and prefixes are resolved, and the tokens are then transformed into some representation of the document Infoset; at the same time, checking for well-formedness[1,2] is performed. The Infoset is optionally checked against an XML Schema grammar (XML schema, schema) for validity and rendered to the user through some interface, such as Simple API for XML (SAX) or Document Object Model (DOM) (API stands for application programming interface).

In practice, these tasks are combined to some extent. Typically a generic parser handles scanning, XML normalization, namespaces, and well-formedness checking, as required by the XML specification. Validation is then grafted on as a separate process, operating as a filter on the output of the generic parser. Because validation is an add-on in such a design, it has a strictly detrimental effect on parser performance. Validation is, therefore, typically used exclusively for debugging, if at all, and is disabled during production.

Although schema validation is expensive in the above scenario, there is no fundamental reason why validation needs to be expensive. Indeed, grammars have long been used to generate optimized special-purpose parsers that operate much more efficiently than their generic counterparts, while performing validation checking.[6] The XML specifications were designed to enable the compilation of an XML Schema grammar to a special-purpose parser (see Section 2.4 in Reference 5).

Techniques that apply standard grammar-based parser generation to XML Schema grammars have been used to demonstrate that compilation of schemas can produce high-performance special-purpose parsers.[7] Traditional parser-generation schemes are not, however, particularly well-suited to XML parsing and have difficulty representing some XML Schema constructs that are not found in traditional parsing situations. At the same time, the syntax of XML and the constraints defined in an XML schema are not sufficiently complex to require the full power of traditional parser-generation methods.

Previous efforts in this area that built on conventional intermediate representations have, in general, supported fewer features of XML[8] or delivered less efficient solutions.[9]

Rather than adapting a conventional intermediate representation to the forms of XML and XML Schema, we propose a compilation technique that deals directly with the abstract schema components of the XML Schema Recommendation.[5] By tying the code-generation scheme directly to the schema components, we are able to take advantage of the simple lexical structure of XML and the determinism assurances built into XML Schema grammars.

The generated validating parser drives the optimized scanning process. Two complementary optimization strategies, specialization and optimistic scanning, are used to speed scanning and validation. Specialization focuses on the use of specialized context-sensitive scanning primitives that can scan and validate the input efficiently. Optimistic scanning speeds the scanning of the common cases, such as simple data without comments or entity references. The resulting generated parser is shown to be significantly faster than some widely available parsers, both validating and nonvalidating.

In the next section, we describe the challenges involved in generating parsers through compilation of XML schemas. Following that, we propose an architecture for direct schema compilation, highlighting the breadth of support for schema features and the targeted optimizations that minimize the cost of parsing. Then we provide performance measurements of sample generated parsers and compare those to performance measurements of commonly used parsers. Finally, we describe related work from the technical literature and conclude with final comments.

CHALLENGES OF XML SCHEMA COMPILATION

XML Schema, and the specifications on which it depends, present several challenges to schema-based parser generation: XML namespaces and the dynamic typing features of XML Schema complicate the scanning of markup. As a result, the schema grammar and the lexical production of XML are not easily combined with traditional grammar compilation techniques. Additionally, XML Schema provides support for content models that are difficult to represent in traditional automaton models, making the traditional models inefficient as intermediate representations of the schema.

XML namespaces

Throughout the XML processing stack, markup and meta-markup (such as namespace declarations and `xsi:type` attributes) assert scoped properties and declarations for the containing element and all of its attributes, as well as its content. In the case of XML namespaces and the dynamic typing mechanism used in XML Schema, this pattern presents some difficulties for naïve parser implementations.

Namespaces qualify XML element and attribute names by using a namespace-prefix declaration. The declaration takes the form of a special attribute with a reserved prefix (`xmlns`) followed by the prefix to

be declared. The value of this attribute is the declared namespace. The scope of the namespace declaration includes the enclosing element, all of the sibling attributes, and the element's content. This arrangement, although natural, presents some difficulties for XML processors.

Beyond the syntactic complexities of the namespace declaration, the XML Namespaces Recommendation augments the well-formedness constraints of XML to forbid the repetition of a qualified attribute regardless of its lexical representation in the tag. Thus, in the examples below, the first `my-elt-1` is well-formed, but `my-elt-2` is not, because both attributes resolve to the same qualified name.

```
<a:my-elt-1 a:my-attr="true" b:my-attr="false"
            xmlns:a="http://www.example.com/a"
            xmlns:b="http://www.example.com/b"/>

<a:my-elt-2 a:my-attr="true" b:my-attr="false"
            xmlns:a="http://www.example.com/a"
            xmlns:b="http://www.example.com/a"/>

<a:my-elt-1 a:my-attr="true" b:my-attr="false"
            xmlns:a="http://www.example.com/a"
            xmlns:b="http://www.example.com/c"/>
```

During processing, namespace declarations prevent the qualified names of the element and its attributes from being conclusively known until the end of the tag. This means that scanning of qualified names in XML requires infinite look-ahead to fully resolve names. In the preceding examples, the second `a:my-elt-1` appears to be the same as the first until the last attribute is scanned.

The pattern of declaration used in XML namespaces is typical throughout the XML processing stack. In the XML layer, for example, the predefined attributes `xml:lang` and `xml:space`, which may be used to indicate natural language and desired white-space handling, use this pattern. The values of these attributes, however, do not affect validation, and therefore do not complicate scanning or validation.

Dynamic typing in XML Schema

XML Schema includes a mechanism for dynamic typing of instance elements. By using the `xsi:type` attribute, an instance element may assert its XML Schema type. The declared type must be validly derived from the type that would otherwise have been used to validate the element, with respect to the constraints on type derivation. This declared type may have a significantly different content model from the default type that is otherwise expected.

The syntax of `xsi:type` is similar to that of namespace declarations and poses the same kinds of processing hurdles. In particular, the possibility of an `xsi:type` attribute prevents an XML processor from conclusively determining the type declaration to use for validation until the entire tag has been scanned. Furthermore, because the element type declaration governs the type declarations used to validate the attributes, the processor cannot conclusively determine the types—and therefore the validity—of the attributes until the entire tag has been read. In the example below, the element will be invalid if the dynamic type restricts the attributes to be of type `xsd:integer`:

```
<my-elt-1 my-attr-1="one" my-attr-2="two"
          my-attr-3="three" my-attr-4="four"
          my-attr-5="five" my-attr-6="six"
          xsi:type="six-integer-attributes"
          xmlns:xsi="http://www.w3.org
            /2001/XMLSchema-instance" />
```

XML Schema content models

Like the Document Type Definition (DTD) grammar used in XML, XML Schema can specify an element's content model as a regular expression over its contained element. In contrast to the grammars that can be specified with an XML DTD, however, XML Schema supports a wider range of operators in the composition of content models. In particular, the arbitrary finite occurrence constraints and `xsd:all` groups of XML Schema pose challenges to automaton-based approaches to compilation. Arbitrary finite occurrence constraints can lead to an explosive growth in the number of states for simple automaton approaches. In the standard implementation, an element declaration with a maximum occurrence constraint of 5000 will result in 5000 states corresponding to each possible occurrence in the range. Models of `xsd:all` group content are not represented in any standard regular expression syntax and require significant augmentation of the automaton model. If translated directly into a standard automaton model, `xsd:all` groups result in an expansion of states that is combinatorial in the number of members of the group.

Because the xsd:all compositor is not well represented in traditional models, much of the previous work on XML schema compilation has treated xsd:all groups as a "corner case" (that is, unimportant). In practice, however, xsd:all groups are quite important, as the xsd:all group is considered to be a natural translation of a data structure with named fields, such as a C struct or a Java class, where the members are identified by name, rather than by position. In practice, we have seen that XML schemas for data stored in relational databases often have a plethora of xsd:all groups. These scenarios are quite common for Web services.

XML character data

In addition to the particular challenges posed by the various specifications involved in XML parsing and validation, the layering of the specifications presents challenges of its own. In particular, the constraints imposed by an XML schema operate on an abstract representation of an actual XML instance described in the Infoset as an abstract tree of information items.[4] All data in the Infoset is represented in a fully expanded form, with entities and character references expanded, CDATA sections replaced with their contents, and end-of-line and attribute normalization completed, as required by XML. This means that the lexical productions and value constraints used by XML Schema to constrain data are defined in terms of this abstract form. As a result, these constraints are typically implemented in a two-pass method, where the content is first scanned according to the lexical-level productions of XML and then normalized and validated against its constraining type. This procedure is inefficient because it requires the data to be scanned twice.

ARCHITECTURE FOR DIRECT SCHEMA COMPILATION

We present now an architecture for direct compilation of XML schemas in which the code for parsing and validation is generated directly from the schema components defined by the XML Schema Recommendation. We describe how this approach simplifies support for schema features that do not translate well to other grammar representations. We also show that the simplified custom-compilation model enables a variety of targeted optimizations that maximize scanning performance without the complications normally associated with combined compilation of XML Schema grammars and XML concrete syntax. In the following sections, we first describe the overall design of the compiler and the generated parser. The algorithms used in the compilation engine are then presented, followed by the design of the generated parser.

Design overview

As discussed in the previous section, from a validation standpoint, the structure of an XML document constrained by a schema cannot be decomposed below the tag level. Because metamarkup, such as namespace and xsi:type declarations, is contained in conceptually unordered attributes, no conclusive information about the document can be inferred until the entire tag is read. Thus, no exchange of information between the scanner and the validation logic can be made to refine the scanning of the rest of the tag without possibly having to back up and correct mistaken assumptions. As a result, the grammar must not direct scanning at a granularity any finer than the tag. Accordingly, the generated validation logic may be cleanly separated from the scanning infrastructure at the tag level, without loss of any significant performance opportunity. Thus, we divide the generated parser into two logical layers, scanning and validation.

The validation layer is a generated recursive-descent parser that drives the scanner by using compiled, predictive knowledge from the schema. The scanning layer consists of a set of fixed XML primitives that scan content at the byte level, at the direction of the validation layer.

The validation logic is produced directly from the schema component model, using component-specific code templates for the various components in the schema. This approach is enabled by a constraint on valid schemas ensuring that all content models are deterministic. This constraint is called the Unique Particle Attribution (UPA) constraint, and is defined in Section 4.4 of the specification as follows[5]:

> A content model must be formed such that during validation of an element information item sequence, the particle component contained directly, indirectly, or implicitly therein with which to attempt to validate each item in the sequence in turn can be uniquely determined without examining the content or attributes of

that item, and without any information about the items in the remainder of the sequence.

This constraint ensures that transitions between particles in the content model are deterministic. The implications of the constraint are somewhat complex and often misunderstood. We will therefore briefly consider a few informative examples using the compact DTD-style syntax for the grammars. The first example below is legal, because as soon as the parser encounters the first element (an `A`, `B`, or `C`) it knows which branch of the disjunction to validate against, `AA`, `BA`, or `CA`, respectively.

```
AA | BA | CA
```

On the other hand, if all three branches begin with the same symbol, as in the next example, the content model is not considered deterministic:

```
AA | AB | AC
```

This is because after encountering the first element (an `A`), the parser does not know which `A` to validate against. In the language of the Schema Recommendation, it does not know which *particle* to assign to the `A` element in the instance: the one in the first branch, `AA`, the one in the second branch, `AB`, or the one in the third branch, `AC`. Because XML Schema allows arbitrary annotations to be attached to each particle in the schema, significant differences may exist between these apparently identical particles.

Assuming that all three `A` particles are identical from the point of view of the schema author, the problem can be remedied easily be rewriting the illegal schema grammar into a legal one. The key difference is that there is only one `A` particle—so there is no ambiguity about which particle validates an `A` in the instance:

```
A (A | B | C)
```

The determinism ensured by the UPA constraint enables particles to be compiled individually. Because any particle that validates a particular input element within a given context is necessarily the only particle that will do so, the validation code generated for the particles in a schema can be composed together without any look-ahead and without requiring backtracking. This is easily represented with greedy logic but could equivalently be represented with non-greedy logic because the two are necessarily equivalent. In any grammar that conforms to the specification, therefore, the name of the most recently read tag is sufficient to determine which branch of the program to execute. This allows code to be generated directly from the schema components, while maintaining the assurance that it will be sufficiently general for all unambiguous schema grammars. Thus, the particles and their terms form a sufficient basis for validation-logic compilation. In the next section, the procedure for generating the validation logic is explained in detail. Deterministic but ambiguous content models will be discussed later.

Schema compilation

The compilation procedure takes place in three stages. The input schema is first read and modeled in terms of abstract schema components (see Section 3 in Reference 5). The complete schema is then augmented with a set of *derived* (calculated) components and properties used to drive code generation. Finally, the schema is traversed, in a recursive-descent fashion, to generate the validation code for each component.

In the following subsections we describe the augmented set of schema components and the derived properties that supplement the component model with information needed for compilation. Then we discuss the process for generating validation code from our code templates and show the templates associated with each schema component. Finally, we provide an example that shows how the stand-alone templates are composed to generate a validating parser.

Schema components

To represent and operate on the XML Schema grammar, we use a publicly available implementation of the schema components. The schema components, taken in aggregate, are referred to as the *schema*. It is assumed that the schema for any given grammar is fully resolved before compilation begins; that is, there are no missing subcomponents, and no attempt is made to further resolve components. The justification of this assumption is provided by the Schema Recommendation itself. ***Figure 1*** shows the relevant schema components, with their compilation annotations in italics, as well as two special components (*synthetic-element* and *skip-term*, described later).

The schema components have four primary component types: element declarations, attribute decla-

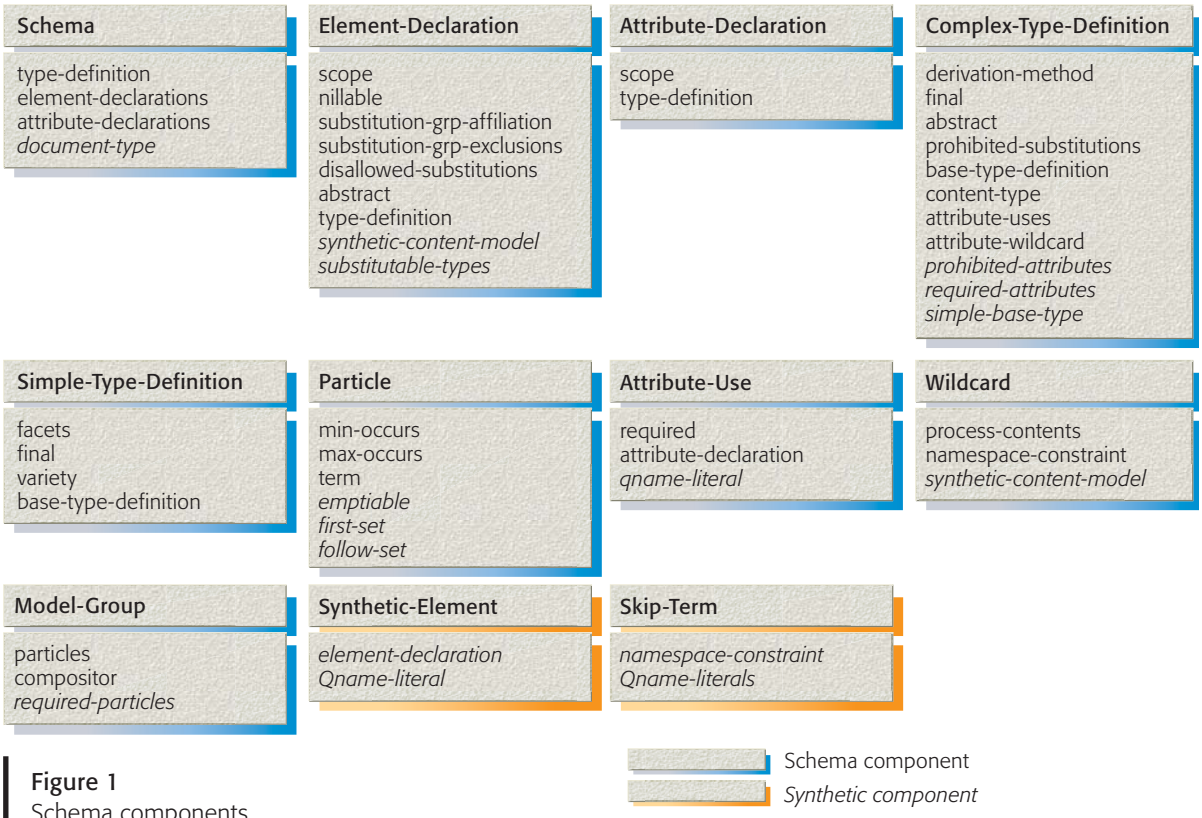

Figure 1
Schema components

rations, complex type definitions, and simple type definitions. Complex type definitions also reference a set of helper components: particle, model group, wildcard, and attribute use.

Complex types may have content that is simple, complex, or empty. When the content is simple, the value of the content-type property is a simple-type definition that defines the content. When the content is empty, the content type is empty. If the complex type has complex content, then the content-type is a particle, which defines a complex content model. The content model for such a complex type is defined in terms of the helper components (particles, model groups, and wildcards). Particles and model groups structure the content model for validating element content, which is eventually validated by element declarations or wildcards. The basic unit of the content model is the particle. See the XML Schema Recommendation:

> A particle is a term in the grammar for element content, consisting of either an element declaration, a wildcard, or a model group, together with occurrence constraints. Particles contribute to validation as part of complex type definition validation, when they allow anywhere from zero to many element information items or sequences thereof, depending on their contents and occurrence constraints.

A particle has a pair of occurrence constraints, `min-occurs` and `max-occurs`, and a `term`. The `term` of a particle can be an element declaration, a model group, or a wildcard. Model groups, in turn, compose groups of particles, using one of three composition models (`xsd:sequence`, `xsd:choice`, `xsd:all`). These components can be combined freely, within the constraints of the UPA constraint, as discussed earlier. Note that in order to facilitate processing, the XML Schema Recommendation places extra restrictions on the use of model groups with the `xsd:all` compositor.

Synthesis of implicit content models

Because of the open-ended composition model of XML Schema, the schema components as defined by the specification lack explicit representations of validation constraints that reference the schema globally. In particular, the content model for wild-

cards, element substitution groups, and the content model for the document itself all make implicit references to the global element declarations of the schema without enumerating them. In the compiler, these implicit validation rules are rendered explicitly with content models synthesized from the standard schema components and the global properties of the schema. These *synthetic content models* are represented with the normal schema components, and with two additional *synthetic components*. The definitions for these components are included in Figure 1.

Document content model. In contrast to a DTD, XML Schema provides no way to indicate the content model for the document itself. The validation rule for the document element is normally taken to be similar to that of a wildcard, matching any global element. To represent this, we define a virtual top-level type for the content of the document. This top-level type is a complex type called `documentType`, and is defined within a private namespace (http://www.ibm.com/XML/impl/types). Unless otherwise stipulated, the `documentType` is assumed to take a form similar to that of `xsd:anyType`, but somewhat more restrictive in that it bears no attributes, forces strict processing, and does not allow mixed content (thus emulating the production of an XML document, as described in Section 2.1 of the XML specification[1,8]). In the XML representation of the schema components, this can be written as:

```
<xsd:schema targetNamespace=
  "http://www.ibm.com/XML/impl/types"
            xmlns:xsd="http://www.w3.org/
              2001/XMLSchema" >
  <xsd:complexType name="documentType" >
    <xsd:sequence>
      <xsd:any/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Element and wildcard content models. Element substitution groups allow for the substitution of one named element for another. Any global element declaration may serve as the head of a substitution group, and any element with a properly derived type may declare itself to be substitutable for the head element.

In a fixed schema, the validation rule for an element substitution group acts as a choice over the

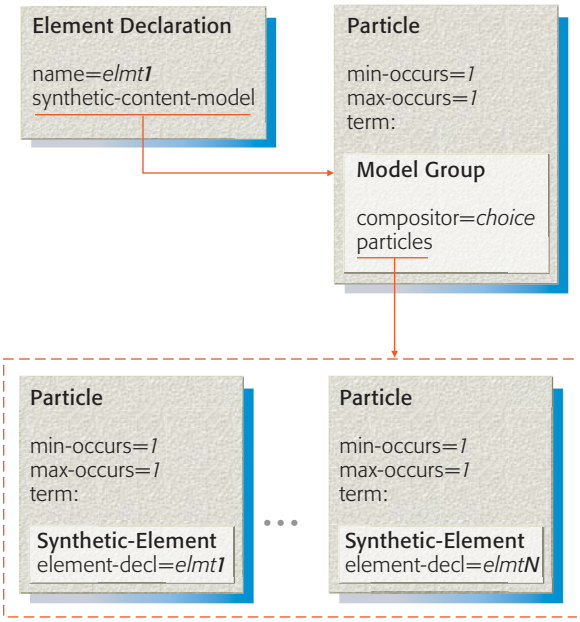

Figure 2
Element synthetic-content model

appropriate element declarations. To represent this explicitly in the compiler, we augment the element declaration component with a *synthetic-content-model* property that represents the expanded form of the element declaration (***Figure 2***). This expanded form was implicitly part of the original schema component. The new content model is a choice over any elements that could transitively appear in the substitution group headed by this element. In order to distinguish an element declaration (which is always considered to be the head of a substitution group) from the terms of this synthetic choice, we define a synthetic component called the *Synthetic-Element*, which like regular element declarations, wildcards, and model groups may be the term of any particle. The Synthetic Element, as opposed to the regular element declaration, validates only the declared element and not any of its substitution group members.

The content model for wildcards is similarly implicit (***Figure 3***). The structure of the validation rule for a wildcard depends on the value of its process-contents property. When *skip* processing is stipulated, the processor is required to *skip* over the matching element and all of its content without any validation. When *strict* processing is stipulated, the matching element must be validated with one of the global element declarations in the schema. *Lax*

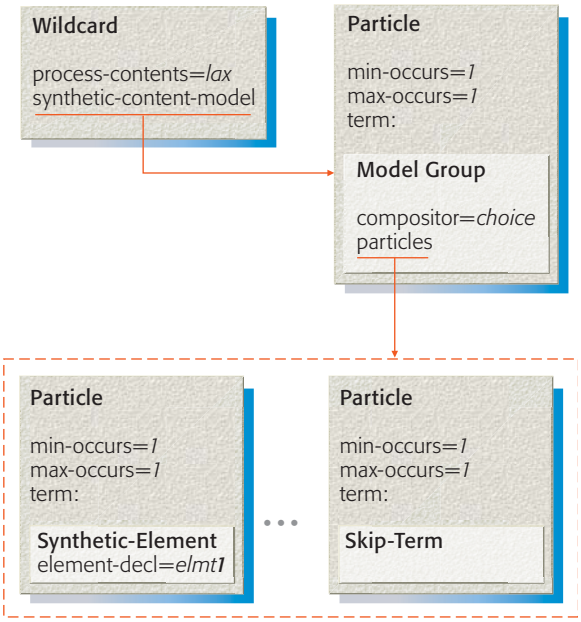

Figure 3
Wildcard synthetic-content model

processing combines the two options, requiring full validation of known elements, but allowing *skip* processing for unknown elements. In all cases, the matching element must satisfy the namespace constraint specified on the wildcard component.

In the compiler, the validation rule for a wildcard is represented with a choice similar to that used for element substitution groups. This is assigned to the *synthetic-content-model* property of the wildcard. Skip processing is represented with a special synthetic component, the *Skip-Term*. If the process-contents property is *strict* or *lax*, the choice contains a particle with a Synthetic-Element term for each global element declaration that satisfies the namespace constraint. If the process-contents property is *skip* or *lax*, the model-group also contains a particle with a Skip-Term.

Derived properties of schema components

Once the schema is fully resolved, the derived properties may be computed. These properties form the basis for the code generation phase that follows. Because several of the properties represent global information about the schema, such as the complete set of global elements matching a wildcard component, the derived properties must all be computed before code generation begins. The properties must be calculated in order because the calculations of one step are used in subsequent steps. Note that the calculation procedures implicitly assume that the schema is valid with respect to all of the constraints on schema components in the XML Schema Recommendation.

Substitutable types. The *substitutable-types* property of an element declaration defines the set of types that can appear in the instance document by using the `xsi:type` dynamic typing mechanism, instead of the declared type. The substitutable-types set contains all global types, including the declared type itself, that possess the following characteristics: they are not anonymous or abstract; they are transitively derived from the declared type; and they do not, at any step of derivation, violate the prohibited-substitutions properties of the element declaration and type definition. In the generated parser, the substitutable-types set is used to validate the value of any `xsi:type` attributes that may appear in the document.

QName-literals. In validating an XML document, a correspondence is made between literal element and attribute names and QNames found in the schema. To make this correspondence, we define a symbol called the *QName-literal*. A QName-literal symbol may represent a specific QName referenced in the schema, or some unbounded set of QNames not directly referenced in the schema but indirectly referenced by a wildcard. Additionally, special QName-literal symbols are used for the close-tag and the end-of-file symbols. At the abstract level, validation constructs are considered to validate sets of QName-literal symbols in the case of attributes, or sequences of QName-literal symbols in the case of content models.

A QName-literal symbol can have one of several forms, as shown in ***Figure 4***:

- A QName explicitly referenced in the schema is represented by the *known* QName-literal symbol, which is the {URI, local-part} pair.
- An unknown QName in a known namespace is represented by a *namespace-known* QName-literal symbol, with a single property, the URI (Uniform Resource Identifier).
- QNames with an unknown namespace are represented by the special singleton *unknown* QName-literal symbol, regardless of their "local-part."
- Close tags, regardless of their name, are all represented by the special *close* QName-literal

symbol. Similarly, the end of file is represented by the *eof* QName-literal symbol.

Note that the set of known QName-literal symbols is considered to be established completely before compilation begins, and that unknown and namespace-known QName-literal symbols are not used to refer to known QNames. Thus, an element wildcard may validate known, namespace-known, and unknown QName-literal symbols.

Particles. Every particle in the schema has three calculated properties: `emptiable`, `first-set`, and `follow-set`.[10] These properties define the relationship between the schema component and the QName-literal sequence it will validate.

The `emptiable` property corresponds to the Particle Emptiable definition is Section 3.9.6 of the XML Schema Recommendation[5] and determines whether the particle can validate the empty QName sequence. The `emptiable` property is used to calculate the other particle properties described next.

The `first-set` property of a particle defines the set of QName-literal symbols that can occur in the first position of an element QName-literal sequence that is validated by the particle. The first set is used to build control-flow logic for the content model; as a direct result of the UPA constraint, comparison of an input QName-literal symbol to the calculated `first-set` property of a particle immediately determines whether or not that particle validates the input sequence. The `first-set` property is calculated recursively, in a single pass over the schema.

The `follow-set` property of a particle defines the set of QName-literal symbols that can follow a QName-literal sequence validated by that particle. The `follow-set` property is used to drive context-sensitive tag scanning. After the `first-set` property is calculated for every particle, the `follow-set` property is calculated in a second pass over the schema components, using the `first-set` properties of adjacent components.

Attribute occurrence constraint validation. For each complex type we calculate two sets of attribute QName-literal symbols, *required* and *prohibited*. These sets are used to validate the attribute occurrence constraints. The required set includes the attribute QName-literal symbols that are required to appear in the input tag. The prohibited set includes the attribute QName-literal symbols that are not allowed to appear in the input tag.

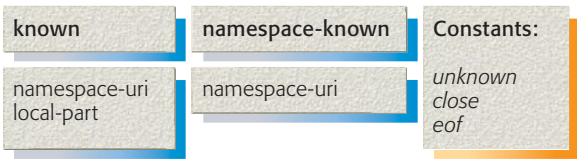

Figure 4
QName-literal symbols

The required and prohibited sets are created from information in the attribute uses of the complex type. Entries for attribute wildcards are also included, in the form of known QName-literal symbols, namespace-known QName-literal symbols, and the unknown QName-literal symbol, depending on the wildcard's process-contents value.

Generation of validation logic

The generated parser consists of modules validating each type in the input schema, including the synthetic `documentType` (see "Document content model" in the subsection "Synthesis of implicit content models" in the previous section). The validation logic is produced directly from the schema component-model representation of each type. Validation code for simple types is largely independent of the input schema, and in our prototype implementation consists mostly of library code. The simple type validation code is described in the section on scanning infrastructure later.

For every complex type we define a recursive-descent *parse* function that parses all the attributes and content of the complex type. To validate element content in a complex type, we also define the element *dispatch* function. This function handles element-specific validation constructs, such as defaulting and nilability, as well as dynamic typing, and dispatches a call to the actual type's parse function. Together, the type parse functions and element dispatch functions make up the whole validation engine.

The main entry point of the generated parser is the parse function for the `documentType`. Starting with the parse function for the `documentType`, control passes back and forth between parse and dispatch

functions, descending through the different types in the schema. The code for each complex type is generated using component-specific templates, described next.

Component templates

For the purposes of this discussion, we ignore out-of-band constraints such as ID/IDREF (Section 3.3.1 in Reference 1) and key/keyref/unique (Section 3.3.2 in Reference 5). Although we did design an implementation for many of these features in order to prove that they would have little effect on the rest of parsing and validation, they are not implemented in our prototype. These constraints are orthogonal to the content model validation code presented and would have no impact on them, if implemented, as well as little noticeable performance impact. We also do not implement a few XML features such as DTD-internal-subset support.

The validation logic for complex content is generated by mapping the various schema components in the content model to code templates. The templates, like the components themselves, are composable. For any component type, there is at least one generic template that will produce validation code for that component. In addition, there may be several optimized templates tailored for common, simple use cases. Using optimized code templates for common use cases helps to minimize the size of the generated code and results in highly optimized validation logic.

Templates for the content model schema components are presented next, in pseudo-C code. In the templates, compile-time substitutions are indicated as follows:

- `COMPILE[x]` marks the insertion of the compiled code for the schema-component specified by *x*, relative to the current schema component.
- `ID[x]` represents a constant for the QName-literal *x*.
- `SET[x]` represents a constant set for the given QName-literal set.
- `SET_CASE[x]` represents a series of switch cases (all with the same body) for each of the QName-literal symbols in *x*.
- `IF[x]` indicates a conditional section of the template that is evaluated at compile time.
- `READ_TAG[x]` is used to mark the insertion of the appropriate read-tag primitive (described later) for the QName-literal set *x*.
- `READ_SIMPLE_CONTENT[x]` marks the insertion of the appropriate read-content function for the simple type specified by *x*.
- `DISPATCH[x]` represents a call to the *dispatch* function to validate the type of the element declaration *x*.

In our prototype, we have implemented all comparisons of QName-literal sets as bit-vector operations. At compile time, the literal bit vectors are calculated, and at runtime, the instance literals are compared against the set in bulk.

Particle. Particle templates handle occurrence constraints. They must also handle emptiability, which interacts with the `min-occurs` property. The generic template is:

```
int count = 0;
label: while (count < max) {
    if (!SET[first-set].contains(current_tag))
        break label;
    COMPILE [term];
    count++;
}
IF [!emptiable] { if (count < min) Fail(); }
```

For an unbounded particle, with `maxOccurs="unbounded"`, there is no upper bound to check, thus the template may be simplified:

```
int count = 0;
label: while (true) {
    if (!SET[first-set].contains(current_tag))
        break label;
    COMPILE[term];
    count++;
}
IF[!emptiable] { if (count < min) Fail(); }
```

An optimized template for optional particles (particles that have `maxOccurs=1`, and are emptiable) can be greatly simplified:

```
if (SET[first-set].contains(current_tag))
   COMPILE[term];
```

The template for fixed-repeat particles (particles that have `minOccurs=maxOccurs` and are not emptiable) is similarly trivial:

```
int count=0;
while (count < max) {
```

```
        COMPILE[term];
        count++;
}
```

The template for trivial particles (particles that have `minOccurs = maxOccurs =1` and are not emptiable) is simply the code for the term.

Model-Group: xsd:sequence. The generic template for model groups with compositor `xsd:sequence` is just a sequence of substitutions. There is no need for optimized templates. In the model-group templates, up to three particles are shown for demonstration. In cases where more (or fewer) particles may exist, `[...]` is used to represent a continuation of the pattern for the rest of the particles.

```
COMPILE[particles.0];
COMPILE[particles.1];
COMPILE[particles.2];
[...]
```

Model-Group: xsd:choice. The generic template for a choice is a simple switch statement:

```
switch (current-tag) {
  SET_CASE
    [particles.1.first-set]:
      COMPILE[particle.1]; break;
  SET_CASE
    [particles.2.first-set]:
      COMPILE[particle.2]; break;
  SET_CASE
    [particles.3.first-set]:
      COMPILE[particle.3]; break;
  [...]
  default: Fail();
}
```

An optimized template for a choice with two particles simplifies the switch to the more efficient if-else clause. A choice with only one particle is a direct substitution of that particle.

Model-Group: xsd:all. All-group templates make use of a set to check occurrence constraints. The set is checked at runtime against the set of required particles. The required-particles set contains an entry for each particle in the model group that has `min-occurs = = 1`. The entries in the set are one-based indexes into the list of particles. The generic template is as follows:

```
Set s;
while (1) {
    switch (current-tag) {
        SET_CASE[particles.1.first-set]:
            if (!s.add(1)) Fail();
            COMPILE[particles.1];
        SET_CASE [particles.2.first-set]:
            if (!s.add(2)) Fail();
            COMPILE[particles.2];
        SET_CASE [particles.3.first-set]:
            if (!s.add(3)) Fail();
            COMPILE[particles.3];
        [...]
            case ID[close]:
                break;
        default: Fail();
    }
}
if (!SET[required-particles].isSubsetOf(s))
   Fail();
```

An optimized all-groups template with no required-particles set does not need the final test for the existence of all required particles. As with choice, an all-group with exactly one particle is a direct substitution of that particle. As with sequence, an all-group with no particles validates the empty sequence. The template is therefore a no-op.

Element Declaration/Wildcard. Validation code for element declarations and wildcards are produced by their synthetic-content models. Synthetic-elements validate exactly one element. Here, the `follow-set` property is the `follow-set` property of the enclosing particle:

```
if (ID[QName-literal] !=current-tag) Fail();
DISPATCH[element-declaration];
READ_TAG[follow-set];
```

The content model for a skip term, which is used by the synthetic-content-model of skip and lax wildcards, repeatedly calls the scanner to scan through one well-formed element. Here, the `first-set` and the `follow-set` properties are those of the enclosing particle.

```
if (!SET[first-set].contains(current_tag))
   Fail();
count=1;
while (count>0) {
    read_tag_mixed();
```

```
    if (ID[close] == current-tag) {
        count--;
    } else {
        count++;
    }
}
READ_TAG[follow-set];
```

Complex type. The template for complex types is composed of a header that handles attributes and `xsi:nil` and a body that handles content. The header template is the same for all complex types:

```
if (!SET[prohibited-attributes].isDisjointFrom
    (current_attributes))
    Fail();
if (!SET[required-attributes].isSubsetOf
    (current_attributes))
    Fail();

if (current_attributes.isPresent(
            ID[attribute-uses.1.QName-literal]))
    COMPILE[attribute-uses.1];
if (current_attributes.isPresent(
            ID[attribute-uses.2.QName-literal]))
    COMPILE[attribute-uses.2];
if (current_attributes.isPresent(
            ID[attribute-uses.3.QName-literal]))
    COMPILE[attribute-uses.3];
[...]
handleNil();
```

The body template for complex types with complex content reads the next tag and calls the code to validate the content-type particle:

```
READ_TAG[particle.firstSet];
COMPILE[particle];
if (ID[close] != current_tag) {
    Fail();
}
```

Note that in the special case of the `documentType`, the template is modified to compare the final tag against `EOF` rather than `CLOSE`.

The body template for complex types with simple content uses `simple-base-type`, which is the complex type's nearest ancestor of simple type:

```
READ_SIMPLE_CONTENT[simple-based-type];
```

The body template for complex types with empty content is simply `read-tag-close`.

Template composition example

We now present an example to illustrate the composition of the component templates. The generated source code is subject to standard well-known source-language optimizations, which are performed by the C-language compiler that is used to compile the generated parser. Some of these optimizations have been applied by hand in the examples below to improve readability. Beyond this clean-up, the code presented is actual code generated by the schema compiler.

The example code is generated from the purchase order schema from the Primer in the XML Schema Recommendation. The schema fragments used in the example are given below:

```
<xsd:complexType name="PurchaseOrderTYpe">
 <xsd:sequence>
  <xsd:element name="shipTo" type="USAddress"/>
  <xsd:element name="billTo" type="USAddress"/>
  <xsd:element ref="comment" minOccurs="0"/>
  <xsd:element name="items" type="Items"/>
 </xsd:sequence>
 <xsd:attributename="orderDate"type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="Items">
  <xsd:sequence>
   <xsd:element name="Item" minOccurs="0"
     maxOccurs="unbounded">
     ...
   </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

The `PurchaseOrderType` example demonstrates parsing and validation of attributes, required children, and optional children. The corresponding generated code is shown in ***Figure 5***.

The sample presented in Figure 5, while partially cleaned from its original form, clearly shows its relation to the templates presented in the previous section. It shows that the direct compilation method outlined in this paper is extremely simple to implement and yet capable of producing nearly optimal validation logic for all schema constructs.

Ambiguous grammars

The template method outlined in the previous sections relies on the determinism of valid schemas

```
void parse_NONE_PURCHASEORDERTYPE() {
  {
    if ((! isDisjointFrom(NONE_PURCHASEORDERTYPE_prohibited,
                          ((BitVector)currentAttributeSet))))
      FAIL_VALIDATION();

    /* Validating optional attribute 'orderDate'.  */
    /* Since the attribute is optional, first check that it is there, */
    /* then validate the contents. */
    if (isPresent(currentAttributeSet, ((int)NONE_ORDERDATE))) {
      validateAttribute_XSD_DATE(NONE_ORDERDATE);
    } else {
      /* if this attribute has a default value, use it */
      /* This attr has no default. */
    }

    /*  Handle xsi:nil  */
    if (isPresent(currentAttributeSet, XSI_NIL)) {
      XSD_BOOLEAN nilValue;
      validateReportedAttribute_XSD_BOOLEAN((&(nilValue)), (XSI_NIL));
      if (nilValue) {
        FINISH_EMPTY();
        return;
      }
    }

    /* Element Only Content */
    PARSE_THIS_TAG("shipTo", NONE, SHIPTO, NONE_SHIPTO);
    if (currentTag != NONE_SHIPTO) FAIL_VALIDATION();

    {
      if (NONE_SHIPTO != currentTag) FAIL_VALIDATION();
      READ_CONTENT_NONE_USADDRESS_subtypes();
      PARSE_THIS_TAG("billTo", 6, NONE, BILLTO, NONE_BILLTO);
    }
    {
      if (NONE_BILLTO != currentTag) FAIL_VALIDATION();
      READ_CONTENT_NONE_USADDRESS_subtypes();
      PARSE_NEXT_TAG();
      if (! ((currentTag == NONE_COMMENT) || (currentTag == NONE_ITEMS)))
        FAIL_VALIDATION();
    }
    if (currentTag == NONE_COMMENT) {
      {
        if (NONE_COMMENT != currentTag) FAIL_VALIDATION();
        READ_CONTENT_XSD_STRING_subtypes();
        PARSE_THIS_TAG("items", NONE, ITEMS, NONE_ITEMS);
      }
    }
    {
      if (NONE_ITEMS != currentTag) FAIL_VALIDATION();
      READ_CONTENT_NONE_ITEMS_subtypes();
      PARSE_NEXT_TAG();
      if (currentTag != CLOSE) FAIL_VALIDATION();

    }
    if (CLOSE != currentTag) FAIL_VALIDATION();
    return;
  }
}
```

Figure 5
Code generated from purchase order schema

to ensure that the implied greedy matching will correctly validate the input. In rare cases, however, the determinism ensured by the UPA constraint is not sufficient. In particular, when there is ambiguity between different iterations of the same particle, the constraint is still satisfied, but the naïve validation logic is incorrect. In the example below, the implicitly greedy algorithm in the preceding sample code templates fails to validate a sequence of six `A` elements:

```
<xsd:sequence minOccurs="1" maxOccurs="2">
  <xsd:element name="A" minOccurs="3"
    maxOccurs="4" />
</xsd:sequence>
```

The template can be adapted, however, to produce the correct result by relaxing the constraint and checking the sequence when it is complete against the aggregate minimum, maximum, and any interior prohibited sequences (in this case, a sequence of five `A`s). This can be done because the ambiguity is confined to the aggregate occurrence constraint, and because the validation logic is a direct analogue of the schema component model. As such, the UPA constraint ensures that the code which is directly associated with a particular schema particle should be used to validate given elements in the document, and that this can be uniquely determined without look-ahead.

The problematic case may be generalized to content models of the following form, where i, j, k, l, and m represent nonzero occurrence constraints, with $0 < (j\text{-}i)$ and $0 < i$, and the group references are used to denote arbitrary content models, with the reference to the optional `a` being non-emptiable.

```
<xsd:sequence minOccurs="1" maxOccurs="m" >
  <xsd:group ref="a" minOccurs="i"
    maxOccurs="j" />
  <xsd:group ref="b" minOccurs="0"
    maxOccurs="k" />
</xsd:sequence>
```

The generic case can be seen to have a finite, known aggregate minimum and a known aggregate maximum. In addition, within this range there is a finite and potentially empty set of prohibited interior sequences. This set is finite even when the aggregate maximum is unbounded.

The use of `xsd:group` in two places and variables i, j, k, l, and m has enabled us to construct this generalized example, which includes absolutely every case that can express ambiguity in a legal schema. Generating code as describe earlier, checking against the aggregate minimum, maximum, and any interior prohibited sequences after the sequence is complete, allows us to successfully generate templated code for the ambiguous grammar, and thus for every legal schema defined by XML Schema.

It should be noted that the ambiguous grammars described here are a rare corner case encountered only when finite occurrence constraints are combined with nested content models that end with optional content. This is in contrast to strictly ambiguous, but non-problematic cases allowed by the XML DTD (such as repeated emptiable particles), which do not have the expressive power to define these more broadly ambiguous grammars. As far as we know to date, no one has reported a real-world example of a content model in this problematic class to the XML Schema Working Group.

Grammar-directed scanning

In keeping with the tag-level separation described earlier, communication of data from the scanning layer to the validation layer is made through shared state representing the most recently read tag, and the comments, processing instructions, and character data preceding that tag. Here, and throughout this section, we define *tag* to mean any open or close tag, including all of its attributes. Communication from the validation layer to the scanner is made through direct calls to the scanning primitives, advancing the state forward through the next tag. At any point in the validation logic, the scanner is always positioned immediately after a tag.

The interface between these layers is designed to maximize the ability of the validation logic to drive optimized scanning, leveraging the full power of context sensitivity in the generated code, while minimizing the amount of generated code produced for any given schema. Performance in the scanner is achieved through a number of grammar-based optimizations. In this section we describe, through examples, the range of optimizations that are used.

Optimization strategies

The various scanning primitives that underpin the generated parser address byte-level scanning performance optimizations that can be characterized as

deriving from two complementary strategies: specialization and optimistic scanning.

The *specialization* strategy leverages the grammar context to make use of specialized scanning for known content. For example, when the validation logic directs the scanner to read the next tag, it does so knowing the set of expected QName-literal symbols, based on the `follow-set` property of the preceding particle. In the case where the `follow-set` property contains only the special *close* QName-literal symbol, specialized scanning logic is used that looks for the start of a close tag, followed by the exact bytes already seen in the corresponding start tag. This is significantly simpler and more efficient than the general production for a tag.

One basic example of specialization, *native-encoding validation*, is so pervasive in the scanner that it deserves special mention. Nearly every scanning activity, from scanning angle brackets and white space to basic simple content, is carried out in the native encoding. Thus, when scanning for an angle bracket or digits in an integer, or comparing against a known QName-literal symbol, the input bytes are not transcoded, but rather compared to pre-encoded literals.

Output transcoding is avoided through the same mechanism. Rather than converting input bytes for use by the application, such as in the UTF-16 based SAX API, pre-encoded output values are used wherever possible.

The *optimistic scanning* strategy further leverages context sensitivity by optimistically favoring the common, simple case. This strategy is particularly powerful for simple types, where the common case is quite simple and where complex lexical constructs like comments, processing-instructions, and the ampersand escapes that introduce XML character and entity references are rare. For example, ignoring nonstandard usage, the production for an `xsd:integer` is simply an optional minus sign followed by any number of decimal digits, as shown in the content for integer-element below.

```
<integer-element>123456</integer-element>
```

In contrast, because comments and processing-instructions do not contribute to the validation, and because escapes are resolved before validation occurs, the following is a valid, if perverse, representation of the same number.

```
<integer-element>12<!-- -->34<?x ?>5&#54;
</integer-element>
```

Obviously, scanning the former example is much simpler and can be written quite efficiently, whereas scanning the latter involves a great deal of overhead that will almost never be used in the integer simple type context. The character entity presents particular problems because it must be translated and then validated as a digit, rather than simply ignored.

A pervasive use of optimistic scanning is made in the handling of unknown names, such as wildcard elements and prefixes, in the instance. On the assumption that the parser will be used to parse many instances that use the same prefixes and element names, the scanner saves both the input and output encodings for unknown names in its name table. The well-formedness of an unknown name is checked only when it is added to the table. When reused, it is already known to be well-formed. This means that the scanner rarely has to resort to the full production of a well-formed XML name.

Fast scanning primitives

All of the scanning primitives obey a single contract, defined by the validator-scanner interface. When invoked, each primitive scans from the current scanner offset through the end of the next tag, populating the current shared state.

Read-tag. The basic scanning primitive for non-simple data is read-tag. Following the strategy of specialization, the read-tag primitive comes in several flavors: read-tag, read-tag-one, and read-tag-close. These correspond to the generic case, and the common simple cases of exactly one known QName-literal symbol and exactly the close QName-literal symbol.

The element-name scanning of the read-tag-one primitive is a good example of how the strategies of specialization and optimistic scanning are applied. Rather than scanning the input bytes for a well-formed tag name and then comparing against the expected fixed value, the scanner can use the fixed value to scan through the tag name. Although not implemented in the prototype, another common case that might be optimized easily is read-tag-one-or-close, reading exactly one known QName-literal

symbol or the close literal. This primitive is suitably common, occurring in any scenario where a complex type ends with a repeated or optional element.

Optimistic scanning also comes into play when namespace-qualified tags are read. The scanner first tries to compare the input bytes against the unprefixed name, assuming that the appropriate namespace has been defaulted. In the case where it has not, this comparison fails very quickly, and the check for a prefix can proceed. As with all names, the prefix need not be checked for well-formedness, and thus the scanner can simply scan ahead for the expected colon. Once the colon is encountered, the scanner can then retry the literal name comparison as before.

Note that all of the operations required to scan the tag name make use of basic byte-scanning primitives like `strchr` and `strcmp`. Because these operations are invariably much faster than table-driven scanning on most architectures, read-tag-one can be seen to be significantly faster than generic nonvalidating parsing allows.

Simple-content scanning. Simple-content scanning is handled by a library of simple-type validators. Each validator is specialized to scan an individual built-in simple type or a related set of simple types. As with the read-tag primitives, all of the simple-type scanners scan forward through the next tag. In the case of simple content, this tag is always a close tag, which is scanned with the same mechanism as read-tag-close.

Simple content represents a very special case of XML character data. For most types, it is a safe guess that the data will not contain comments, processing-instructions, or even entity references, as in the `<integer-element>` example earlier. For this reason, optimistic scanning can greatly improve simple-content scanning. To facilitate optimistic scanning, simple-type validators employ a uniform approach to optimistic scanning and fallback. Each scanner implements a `scanOptimistic` and a `validate` routine. The optimistic routine scans through the bytes, matching against the optimistic production. When a byte is encountered that does not fit the optimistic production, the content is rescanned, normalized, and validated with the `validate` routine.

The optimistic routine is called by a standard, boilerplate read-type routine. When the optimistic routine is completed successfully, the main read-type routine looks ahead two characters for the expected close tag and reads it if it is found. If the close tag is not found, or if the optimistic routine is not completed successfully, read-tag-close is used to scan from the original start through the end of the tag. The resulting character data section is then collated according to its white-space facet and validated with the `validate` routine. The code for an example read-type routine is shown below:

```
void readType() {
    final int offset=offset();
    if (scanOptimistic() == OK &&
        peekChar(0) == '<' &&
        peekChar(1) == '/') {
incrOffset(2);
finishClose();
    } else {
setOffset(offset); /* back up */
readTagClose(MIXED, COLLAPSE);
validate();
    }
    checkFacets();
}
```

Optimistic scanners may choose to ignore a wide variety of rare constructs, from comments and processing-instructions, to type-specific issues like leading zeros. The type definitions themselves[11] provide some good initial guidelines for what normal simple data looks like. In addition to the lexical form, the recommendation also specifies a canonical lexical representation, which is a limitation of the general lexical representation, such that each value in the value space has exactly one lexical representation. For many types, the canonical representation provides suitable guidance for an optimistic scanner. For simple content values that conform to the optimistic form, the optimistic scanner is clearly much more efficient than the generic alternative, which has to check for leading or trailing white space, comments, and other content that does not often appear in simple type content.

Even user-defined simple types, which are defined in the schema, can benefit from this type of optimistic processing. In schemas, most user-defined simple types are merely versions of the

built-in simple types with a set of facets applied. Thus, scanning functions like those described can be called for the inherited built-in simple type, and then the facets applied after parsing and validation. This requires storing the value, but that is normally required anyway for reporting in an API.

Attribute scanning and validation. Attributes are naturally unordered, and their scanning is complicated by the look-ahead issues described earlier: XML namespaces, dynamic typing, and so forth. As such, attribute scanning cannot benefit from the same aggressive optimizations as element-name scanning. Grammar knowledge can, however, be used to optimize the evaluation of attribute occurrence constraints. In particular, the set of known QName-literal symbols is used to optimize set operations, such as subset-of and is-disjoint-from, by representing attribute sets as bit vectors with an entry for each known attribute QName-literal symbol.

PERFORMANCE ANALYSIS

We developed a compiler for generating a schema-validating XML parser by using the compilation techniques outlined in this paper. Whereas care was taken to include all features that might have a serious impact on performance and design, such as namespaces, dynamic typing, substitution-groups, simple type validation and all-groups, not all schema features were implemented. In particular, identity constraints (key/keyref/unique) were not implemented, and optimized scanners for many simple types were not implemented. As explained earlier, these would have had at most a negligible impact on performance and complexity if implemented.

Basic performance tests were run comparing the compiled parser against two standard open-source parsers, Xerces 2.6[12] and Expat 1.95.8.[13] Xerces provides a good baseline for performance because it is broadly known, widely available, and widely used for schema validation. We measured Xerces parsing speed in both validating and nonvalidating modes. Expat is generally considered to be a fast XML parser implementation although it does not perform schema validation. In order to manage the overhead of populating an API, both the compiled parser and Xerces render the data as the same SAX-like events, in which all data must be encoded in UTF-16. Expat supports a SAX-like interface as well, except the data is not transcoded.

The tests reported here were run on an IBM eServer* xSeries* Model 235 with a 3.2 GHz Intel Xeon** processor, and 2 GB of main memory, using Microsoft Windows Server** 2003 Service Pack 1. Our parsers were compiled with Microsoft 32-bit C/C++ Optimizing Compiler Version 13.10.3077. All of the parsers were tested on a selection of schema and instance pairs from the Sarvega XML Validation Benchmark[14] and on 1-KB, 8-KB, and 64-KB instances of the purchase-order schema from the *XML Schema Part 0: Primer*.[15] All tested instances were UTF-8 encoded.

Table 1 shows the test cases and numerical results, whereas *Figure 6* shows in bar-graph form a performance comparison between the four XML parsers. The performance is measured in MB/GHz*Sec which represents the throughput in megabytes per second on a 1-GHz processor machine. For 64-KB purchase-order documents, test case 6, the compiled parser is 8.8 times faster than validating Xerces, and nearly four times faster than the non-validating Xerces WFC. The results vary little with document size. Similarly, the compiled parser validates 64-KB purchase-order documents 1.6 times faster than Expat checks the same documents for well-formedness.

RELATED WORK

There have been many different efforts directed at XML parsing performance, including early work into XML formalisms (Murata et al.,[16] Löwe et al.[8]). One theme of these efforts has been to produce variations of deterministic finite automata (DFA), extended in different ways to accommodate the difficulties of XML and XML Schema that we have discussed.

Chiu and Lu[7] extend DFAs to *nondeterministic generalized automata* and describe a technique for translating these into *deterministic generalized automata*, from which a parser can be generated. These parsers operate on a byte level, performing well-formedness checking and validation concurrently, as we do, thus speeding up the parser. Unfortunately, construction of deterministic generalized automata from nondeterministic generalized automata can cause a multiplicative blowup in the number of states. In general, this solution subsets

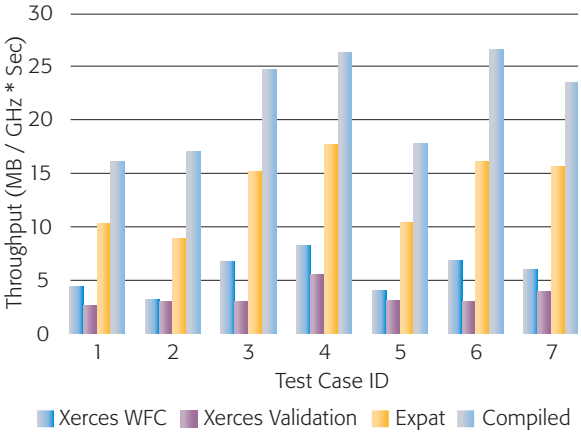

Figure 6
Comparing performance results for four XML parsers

the XML specification and XML Schema Recommendation in many important ways, excluding many commonly used features.

For van Engelen[9] and Reuter and Luttenberger,[17] the limitations of the generalized automata technique led to the use of a two-level approach. Van Engelen[9] used a lower-level FLEX scanner to drive a DFA validation layer. Although the construction of the validation DFA resembles our templated validation code, the separation of the scanning and validating layers prevents scanning optimizations such as using `memcmp` on strings known at compile time and specialized type validators. Van Engelen[9] handles more of the XML specifications than Chiu and Lu[7] but has to run a FLEX dispatch loop for every input character.

Reuter and Luttenberger[17] extend deterministic finite automata with *cardinality constraints* on state transitions that map naturally to the encoding of occurrence constraints. Unfortunately, the CCA does not perform well-formedness checking but runs as a separate layer on top of a separate SAX parser, with the associated performance penalty.

A novel approach to speeding XML parsing is described in Takase et al.[18] The technique obviates the need for compilation but rather relies on parsing a large number of similar XML documents. As documents are read, they are recorded in a DFA, with subsequent documents being compared against the DFA, rather than parsed directly. Where the documents match, cached parsing events are returned. Where the documents differ, the new document is parsed to create new parsing events. This technique relies on byte-level handling of XML instance documents, enabling the types of optimizations described in this paper. However, if the instance documents being compared are semantically identical but not byte-for-byte identical, then spurious parsing will be performed. Furthermore, many of the strings that are truly constant from document to document are declared statically in the schema. In a scenario such as any standard Web service, where the input is constrained by a schema, the overhead in complexity required to store and identify these strings dynamically might be much

Table 1 Test cases and performance results for four XML parsers

			Throughput (MB/Process or GHz*Sec)			
	Test case		Xerces-SAX		Expat	Compiled
ID	Schema Filename	Instance Size (bytes)	WFC	Val	WFC	Val
1	po (on 1kpo.xml)	990	4.41	2.65	10.33	16.12
2	MI_AUS_RESPONSE2_1	1,572	3.21	2.98	8.87	17.00
3	po (on 8kpo.xml)	8,062	6.79	3.01	15.14	24.73
4	bibteXML	8,609	8.28	5.58	17.66	26.25
5	MI_AUS_REQUEST2_1	9,429	4.06	3.16	10.42	17.79
6	po (on 64kpo.xml)	63,754	6.88	3.02	16.13	26.58
7	periodic_table	116,506	6.03	3.99	15.68	23.47

greater than simply compiling them from the schema.

Other attempts to boost performance of XML parsing have focused on changing the form of the XML, such as the various proposals for binary XML forms. Although all of our approaches could be used to validate a binary XML stream, binary efforts inherently lose some of the benefits and flexibility of XML, provide limited speedups, and can be detrimental to interoperability. With speedups as large as we have seen on standard XML streams, it is not clear that the potential performance improvement of binary XML forms justifies the potential cost in interoperability and standardization, the core strength of XML. For a thorough analysis of binary solutions, see Bayardo et al.[19]

CONCLUSION

In this paper we have demonstrated a technique for generating XML parsers in which the compilation of XML Schema grammars starts with the abstract-schema-component model defined in the XML Schema Recommendation. This allows us to benefit from the determinism built into XML Schema, which is inherently reflected in the schema components and results in a simplified compilation engine. Furthermore, this method enables the use of specialized, grammar-sensitive primitives and other forms of specialized and optimistic validation that significantly increase parsing performance without the need for large tables or significant code generation.

The direct schema compilation method allows for simpler code generation than traditional automaton-based models. The simpler model is better suited to the structure and challenges of XML parsing and validation and supports the full expressiveness of XML Schema content models. This includes use of namespaces and dynamic typing, large occurrence constraints, and arbitrary compositions of XML Schema content models.

These features are supported without an explosion in either compile-time states or runtime code size.

Performance of the generated parsers is greatly improved over traditional, interpretive validators. Further, the generated parsers are shown to be significantly faster than even high-performance non-validating parsers.

CITED REFERENCES

1. *Extensible Markup Language (XML) 1.0, Second Edition*, W3C—World Wide Web Consortium (2000), http://www.w3.org/TR/2000/REC-xml-20001006.
2. *XML 1.1*, W3C—World Wide Web Consortium (2004), http://www.w3.org/TR/2004/REC-xml11-20040204.
3. *Namespaces in XML*, W3C—World Wide Web Consortium (1999), http://www.w3.org/TR/1999/REC-xml-names-19990114.
4. *XML Information Set*, W3C—World Wide Web Consortium (2001), http://www.w3.org/TR/2001/WD-xml-infoset-20010316.
5. *XML Schema Part 1: Structures Second Edition*, W3C—World Wide Web Consortium (2004), http://www.w3.org/TR/2004/REC-xmlschema-1-20041028.
6. YACC, Lucent Technologies, http://plan9.bell-labs.com/magic/man2html/1/yacc.
7. K. Chiu and W. Lu, "A Compiler-Based Approach to Schema-Specific XML Parsing," *First International Workshop on High Performance XML Processing*, New York, USA, May 17–22, 2004, ACM Press, New York (2004).
8. W. M. Löwe, M. L. Noga, and T. S. Gaul, "Foundations of Fast Communication via XML," *Annals of Software Engineering* **13**, Nos. 1–4, 357–359 (June 2002).
9. R. van Engelen, "Constructing Finite State Automata for High-Performance XML Web Services," *Proceedings of the International Symposium on Web Services and Applications (ISWS) 2004*, Las Vegas, Nevada, USA, June 21–24, 2004, CSREA Press (2004).
10. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, MA (1985).
11. *XML Schema Part 2: Datatypes Second Edition*, W3C—World Wide Web Consortium (2004), http://www.w3.org/TR/2004/REC-xmlschema-2-20041028.
12. Xerces, Apache Software Foundation, http://xml.apache.org.
13. The Expat XML Parser, Open Source Technology Group, http://expat.sourceforge.net/.
14. XML Validation Benchmark, Sarvega, Inc., http://www.sarvega.com/xml-validation-benchmark.html.
15. *XML Schema Part 0: Primer.*
16. M. Murata, D. Lee, and M. Mani, "Taxonomy of XML Schema Languages Using Formal Language Theory," *Proceedings of Extreme Markup Languages 2001*, Montreal, Quebec, Canada, Aug 12–17, 2001, IDEAlliance Inc., http://www.idealliance.org/papers/extreme03/html/2001/Murata01/EML2001Murata01-toc.html.
17. F. Reuter and N. Luttenberger, "*Cardinality Constraint Automata: A Core Technology for Efficient XML Schema-Aware Parsers*," http://www.swarms.de/publications/cca.pdf.
18. T. Takase, H. Miyashita, T. Suzumura, and M. Tatsubori, "An Adaptive, Fast, and Safe XML Parser Based on Byte

Sequence Memorization," *Proceedings of the 14th International Conference on World Wide Web*, Chiba, Japan, May 10–14, 2005, ACM Press, New York (2005), pp. 692–701.

19. R. J. Bayardo, D. Gruhl, V. Josifovski, and J. Myllymaki, "An Evaluation of Binary XML Encoding Optimizations for Fast Stream Based XML Processing," *Proceedings of the 13th International Conference on World Wide Web*, New York, USA, May 17–22, 2004, ACM Press, New York (2004), pp. 345–354.

Accepted for publication November 29, 2005.
Published online April 27, 2006.

Eric Perkins
IBM Research, 1 Rogers Street, Cambridge, MA 02142 (perkinse@us.ibm.com). Dr. Perkins is a software engineer and has worked at IBM for four years. He received a Ph.D. degree in information technology from Massachusetts Institute of Technology in 2001 for research in algorithms for discrete element simulation. His current research focuses on high-performance XML processing.

Morris Matsa
IBM Research, 1 Rogers Street, Cambridge, Massachusetts 02142 (mmatsa@us.ibm.com). Mr. Matsa has been a software engineer and a researcher at IBM for eight years. In that time, he has developed a number of software prototypes that have been integrated into five different IBM products. He also founded the IBM Extreme Blue internship program. He currently works in the field of high-performance processing of the XML stack. He received B.S. degrees in mathematics and computer science and a Master's degree in computer science, all from the Massachusetts Institute of Technology.

Margaret Gaitatzes Kostoulas
IBM Research, 1 Rogers Street, Cambridge, Massachusetts 02142 (mgg@us.ibm.com). Margaret Kostoulas is a software engineer and has worked at IBM for eight years. She received a Master's degree in computer science from Purdue University. She is currently working on the performance of the XML processing stack. In the past, she worked on Universal Usability projects and on cross-discipline projects, designing virtual prototyping laboratories for computational science.

Abraham Heifets
IBM Research, 1 Rogers Street, Cambridge, Massachusetts 02142 (aheifets@us.ibm.com). Mr. Heifets is a software engineer and has worked at IBM for two years. He received his Bachelor's and Master's degrees from Cornell University. In the past, he has worked on a world champion robotic soccer team, a publish-subscribe system for location-based services, and single-agent search algorithms. He currently works in the field of high-performance processing of the XML stack.

Noah Mendelsohn
IBM Research, 1 Rogers Street, Cambridge, Massachusetts 02142 (noah_mendelsohn@us.ibm.com). Mr. Mendelsohn is a Distinguished Engineer at IBM Research in Cambridge, Massachusetts. He has been working at IBM for over 25 years, has made significant contributions to the development of SOAP, the W3C XML Schema Language, and JavaBeans™, and is a member of the World Wide Web Consortium Technical Architecture Group. He has extensive experience in the areas of operating systems, programming languages, and distributed systems. Mr. Mendelsohn has a Master's degree in computer science from Stanford University and a Bachelor's degree in physics from Massachusetts Institute of Technology. ■