# Taxonomy of XML Schema Languages Using Formal Language Theory

MAKOTO MURATA
IBM Tokyo Research Lab
DONGWON LEE
Penn State University
MURALI MANI
Worcester Polytechnic Institute
and
KOHSUKE KAWAGUCHI
Sun Microsystems

On the basis of regular tree grammars, we present a formal framework for XML schema languages. This framework helps to describe, compare, and implement such schema languages in a rigorous manner. Our main results are as follows: (1) a simple framework to study three classes of tree languages (local, single-type, and regular); (2) classification and comparison of schema languages (DTD, W3C XML Schema, and RELAX NG) based on these classes; (3) efficient document validation algorithms for these classes; and (4) other grammatical concepts and advanced validation algorithms relevant to an XML model (e.g., binarization, derivative-based validation).

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design—*Schema and subschema*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Classes defined by grammars or automata*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: XML, schema, validation, tree automaton, interpretation

## 1. INTRODUCTION

XML [Bray et al. 2004] is a metalanguage for creating markup languages. To create an XML-based markup language, we design a collection of names for elements and attributes that the language uses. These names (i.e., tag names

An earlier version of this article, Murata et al. [2001], was presented at Extreme Markup Language 2001.

Author's address: M. Murata, IBM Tokyo Research Lab, 1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502 Japan; email: EB2MMRT@asahi-net.or.jp.

and attribute names) are then used by application programs dedicated to this type of information. For instance, XHTML [Altheim and McCarron 2000] is such an XML-based language. In it, permissible tag names include `p`, `a`, `ul`, and `li`, and permissible attribute names include `href` and `style`. Then, an application program of XHTML (e.g., XHTML browser) relies on these names for identifying paragraphs, anchors, itemized lists, or links.

An *XML schema* is a rigorous specification of an XML-based language in terms of constraints on elements and attributes. An XML document is said to be *valid* against a schema if the elements and attributes in this XML document satisfy the constraints specified in the schema. For example, a schema for XHTML specifies `p`, `a`, `ul`, or `li` as tag names and specifies `href` or `style` as attribute names. This schema further specifies a constraint that a `ul` element has `li` elements as child elements. The W3C XHTML recommendation [Altheim and McCarron 2000] contains an authoritative schema for XHTML.

We typically use some computer language, which we call *schema language*, for expressing a schema. A *validator* for a schema language is a computer program that determines whether or not a given XML document is valid against a schema written in that schema language. The primary advantage of schema languages is that descriptions in schema languages are more precise than those in prose and that we can rely on validators rather than carrying out human inspections.

Several schema languages have been proposed in the past. Among them, DTD [Bray et al. 2004], W3C XML Schema[1] [Thompson et al. 2001], and RELAX NG [Clark and Murata 2001] are general purpose schema languages created by standardization organizations, and we are mainly concerned about these languages. However, other schema languages are of interest, too. Some (e.g., XDuce [Hosoya and Pierce 2003] and DSD [Klarlund et al. 2000]) are research activities rather than industrial specifications. Others (e.g, RDF Schema [Brickley and Guha 2004]) are special purpose schema languages for a particular type of information which may be represented in the XML syntax. Yet others (e.g., Schematron [Jelliffe 2000]) are languages for representing integrity constraints. We discuss these schema languages in Section 8.

Unlike other survey papers (e.g., Lee and Chu [2000]), we study schema languages using a formal framework. We believe that providing a formal framework is crucial in understanding various features of schema languages and facilitating efficient implementations of schema languages. However, our formal approach has limitations: we do not consider some important characteristics of schema languages such as comprehensiveness, readability, and maintainability. Although these characteristics are highly important, they are outside the scope of this article.

To formally capture schemas, schema languages, and document validation, we use regular tree grammar or tree automata theory [Comon et al. 1997; Takahashi 1975]. Regular tree grammars and tree automata have recently been

[1]Throughout the article, we differentiate two terms, XML schema(s) and W3C XML Schema. The former refers to a general term for schema in an XML model, while the latter refers to one particular kind of XML schema language proposed by W3C in Thompson et al. [2001].

used by many researchers for representing schemas or queries for XML and have become mainstream in this area (see OASIS [2003], Vianu [2001], Neven [2002], Klarlund et al. [2003]). For example, XML Query [Boag et al. 2005] of W3C is based on tree automata.

Our major contributions in this article are as follows.

- Based on three subclasses of regular tree grammars and languages, we classify XML schema languages such as DTD, W3C XML Schema, and RELAX NG.
- We show algorithms for validating documents against schemas under these subclasses and further consider the characteristics of these algorithms (e.g., the tree model vs. the event model, and complexity).
- Based on regular tree grammar theory, we present a detailed analysis and comparison of the XML schema languages with respect to their expressive power and closure properties under boolean set operations such as union, intersection, and difference.
- We also discuss other grammatical concepts (e.g., deterministic content models, balanced context-free grammars) and validation algorithms (e.g., binarization, derivative-based validation) relevant to an XML model and our framework.

### 1.1 Roadmap

The remainder of this article is organized as follows. In Section 2, we introduce the class of regular tree languages, introduce two restricted classes called local and single-type, and study properties of these classes of tree languages. In Section 3, using these classes of tree languages, different XML schema language proposals are analyzed and classified accordingly. In Section 4, we study efficient validation algorithms for these classes. In Section 5, we discuss properties of these classes such as expressive power and boolean closure. Other grammatical concepts relevant to tree grammars and advanced validation algorithms are discussed in Sections 6 and 7. In Section 8, we consider related works such as other survey papers on XML schema languages. Finally, concluding remarks and thoughts on future research directions are discussed in Section 9.

## 2. TREE GRAMMARS

In this section, as a mechanism for describing permissible trees, we introduce tree grammars. Tree grammars generate trees, and as such, they should not be confused with context-free grammars which generate strings. Since XML documents are trees rather than strings, tree grammars are more appropriate than context-free grammars are. We will further compare tree grammars and context-free grammars in Section 5.

We begin with a class of tree grammars called regular, and then introduce two restricted classes called local and single-type.

### 2.1 Regular Tree Grammars and Languages

In preparation, we clarify what we mean by trees. Our trees are ordered (i.e., a node has an ordered sequence of child nodes) and do not have fixed arities (i.e., a node is allowed to have any number of child nodes). Nodes are labeled with the exception of text nodes being leaves. Such trees capture element structures of XML documents.

We borrow the definitions of regular tree languages and tree automata from Comon et al. [1997], but allow the right-hand side of a production rule to have a regular expression over nonterminals.

*Definition* 2.1. A regular tree grammar is a 4-tuple $G = (N, T, S, P)$, where:

—$N$ is a finite set of nonterminals,
—$T$ is a finite set of terminals,
—$S$ is a set of start symbols, where $S$ is a subset of $N$,
—$P$ is the set of production rules of the form $X \rightarrow \mathbf{a}\, r$, where $X \in N$, $\mathbf{a} \in T$, and $r$ is a regular expression over $N$; $X$ is the left-hand side, $\mathbf{a}\, r$ is the right-hand side, and $r$ is called the content model of this production rule.

We often use **bold** lowercase for terminal symbols and capitalized *Italic* font for nonterminal symbols or regular expressions. Furthermore, the null sequence of nonterminals is represented by $\epsilon$. Any text node in a tree is assumed to match a special terminal **pcdata**.

*Example* 2.1. The following grammar $G_{2.1} = (N, T, S, P)$ is a regular tree grammar. The left-hand side, right-hand side, and content model of the first production rule are *Doc*, **doc** (*Para1*, *Para2*$^*$), and (*Para1*, *Para2*$^*$), respectively.

$$\begin{aligned} N &= \{Doc, Para1, Para2, Pcdata\} \\ T &= \{\mathbf{doc}, \mathbf{para}, \mathbf{pcdata}\} \\ S &= \{Doc\} \\ P &= \{Doc \rightarrow \mathbf{doc}(Para1, Para2^*), Para1 \rightarrow \mathbf{para}\ (\epsilon), \\ &\quad\ Para2 \rightarrow \mathbf{para}\ (Pcdata), Pcdata \rightarrow \mathbf{pcdata}\ (\epsilon)\} \end{aligned}$$

Without loss of generality, we can assume that no two production rules have the same nonterminal in the left-hand side and the same terminal in the right-hand side at the same time. If a regular tree grammar contains such production rules, we only have to merge them into a single production rule. For example, $Doc \rightarrow \mathbf{doc}\,(Para1, Para2^*)$ and $Doc \rightarrow \mathbf{doc}\,(Para2^*)$ can be merged into $Doc \rightarrow \mathbf{doc}\,(Para1^?, Para2^*)$. We have to define when a tree is valid against a regular tree grammar. We first define interpretations.

*Definition* 2.2. An interpretation $I$ of a tree $t$ against a regular tree grammar $G$ is a mapping from each node $e$ in $t$ to a non-terminal, denoted $I(e)$, such that:


(a) Tree (b) Interpretation

Fig. 1. A tree $t$ and its interpretation against $G_{2.1}$.

—$I(e)$ is a start symbol when $e$ is the root of $t$, and
—for each node $e$ and its child nodes $e_0, e_1, \ldots, e_m$, there exists a production rule $X \to \mathbf{a}\ r$ in $G$ such that
  —$I(e)$ is $X$,
  —the terminal symbol (label) of $e$ is **a**, and
  —$I(e_0)I(e_1)...I(e_m)$ matches $r$.

Recall that any text node is replaced by the terminal symbol **pcdata**. There will be only one production rule in our grammar that has **pcdata** on the right-hand side, *Pcdata* $\to$ **pcdata**$(\epsilon)$.[2] Now, we are ready to define validity against a regular tree grammar and to introduce regular tree languages.

*Definition* 2.3. A tree $t$ is valid against a regular tree grammar $G$ if there is an interpretation of $t$ against $G$. A set of trees is a regular tree language if, for some regular tree grammar, all trees in this set are valid and no other trees are valid.

*Example* 2.2. A tree <doc><para/><para>text chunk</para></doc> and its interpretation against $G_{2.1}$ are shown in Figure 1. Observe that "text chunk" matches **pcdata** and is associated with *Pcdata*.

## 2.2 Local Tree Grammars and Languages

We first define competition of nonterminals which makes document validation difficult. Then, we introduce a restricted class called local [Takahashi 1975] by prohibiting competition of nonterminals. This class roughly corresponds to DTD.

*Definition* 2.4. Two different nonterminals $A$ and $B$ are said to be competing with each other if

—one production rule has $A$ in the left-hand side,
—another production rule has $B$ in the left-hand side, and
—these two production rules share the same terminal in the right-hand side.

[2] This is a simplifying assumption. The handling of text nodes in real schema languages is much more complicated and is outside the scope of this article.

*Example* 2.3. Consider a regular tree grammar $G_{2.3} = (N, T, S, P)$, where:

$$
\begin{aligned}
N &= \{\mathit{Book}, \mathit{Author1}, \mathit{Author2}, \mathit{Son}, \mathit{Article}, \mathit{Daughter}\} \\
T &= \{\textbf{book}, \textbf{author}, \textbf{son}, \textbf{article}, \textbf{daughter}\} \\
S &= \{\mathit{Book}, \mathit{Article}\} \\
P &= \{\mathit{Book} \rightarrow \textbf{book}\ (\mathit{Author1}), \mathit{Article} \rightarrow \textbf{article}\ (\mathit{Author2}), \\
&\quad \mathit{Author1} \rightarrow \textbf{author}\ (\mathit{Son}), \mathit{Author2} \rightarrow \textbf{author}\ (\mathit{Daughter}), \\
&\quad \mathit{Son} \rightarrow \textbf{son}\ (\epsilon), \mathit{Daughter} \rightarrow \textbf{daughter}\ (\epsilon)\}
\end{aligned}
$$

*Author1* and *Author2* compete with each other, since the production rule for *Author1* and that for *Author2* share the terminal **author** in the right-hand side. There are no other competing nonterminal pairs in this grammar.

The implication of the existence of competing nonterminals in a regular tree grammar is that it makes the validation of XML documents more difficult. That is, given the **author**, validators have to somehow figure out if the **author** matches with either *Author1* or *Author2*.

*Definition* 2.5. A local tree grammar is a regular tree grammar without competing nonterminals. A set of trees is a local tree language if, for some local tree grammar, all trees in this set are valid and no other trees are valid.

*Example* 2.4. The grammar $G_{2.1}$ in Example 2.1 is not local, since nonterminals *Para1* and *Para2* compete with each other.

*Example* 2.5. The grammar $G_{2.3}$ in Example 2.3 is not local, since nonterminals *Author1* and *Author2* compete with each other.

*Example* 2.6. The following grammar $G_{2.6} = (N, T, S, P)$ is a local tree grammar:

$$
\begin{aligned}
N &= \{\mathit{Book}, \mathit{Author1}, \mathit{Son}, \mathit{Pcdata}\} \\
T &= \{\textbf{book}, \textbf{author}, \textbf{son}, \textbf{pcdata}\} \\
S &= \{\mathit{Book}\} \\
P &= \{\mathit{Book} \rightarrow \textbf{book}\ (\mathit{Author1}), \mathit{Author1} \rightarrow \textbf{author}\ (\mathit{Son}), \\
&\quad \mathit{Son} \rightarrow \textbf{son}\ (\mathit{Pcdata}), \mathit{Pcdata} \rightarrow \textbf{pcdata}\ (\epsilon)\}
\end{aligned}
$$

### 2.3 Single-Type Tree Grammars and Languages

Next, we introduce a less restricted class called "single-type" by prohibiting competition of nonterminals within a single content model. This class roughly corresponds to W3C XML Schema.

*Definition* 2.6. A single-type tree grammar is a regular tree grammar such that

- —for each production rule, nonterminals in its content model do not compete with each other, and
- —start symbols do not compete with each other.

Table I. Summary of Examples

| | Regular | Single-Type | Local |
|---|---|---|---|
| $G_{2.1}$ | Yes | No | No |
| $G_{2.3}$ | Yes | Yes | No |
| $G_{2.6}$ | Yes | Yes | Yes |

A set of trees is a single-type tree language if, for some single-type tree grammar, all trees in this set are valid and no other trees are valid.

*Example* 2.7. The grammar $G_{2.1}$ in Example 2.1 is not single-type. Observe that nonterminals *Para1* and *Para2* compete, and they occur in the content model of the production rule for *Doc*.

*Example* 2.8. $G_{2.3}$ in Example 2.3 is a single-type tree grammar; although *Author1* and *Author2* compete with each other, they do not appear in the same content model.

### 2.4 Expressive Power

Having introduced three classes of tree grammars, we devote the rest of this section to properties of these classes. First, we study the expressive power of these classes. This study helps to compare the expressive power of schema languages in Section 3.

By definition, a single-type tree grammar is always a regular tree grammar. Thus, a single-type tree language is always a regular tree language. Likewise, a local tree grammar is always a single-type tree grammar, since the restriction on a local tree grammar is tighter than that on a single-type tree grammar. Thus, a local tree language is always a single-type tree language.

LEMMA 2.1. *Any single-type tree language is a regular tree language, and any local tree language is a single-type tree language.*

We are now interested in the converse: can regular tree grammars be rewritten to single-type tree grammars and can single-tree grammars be rewritten to local tree grammars? The following lemma shows that the answer is negative.

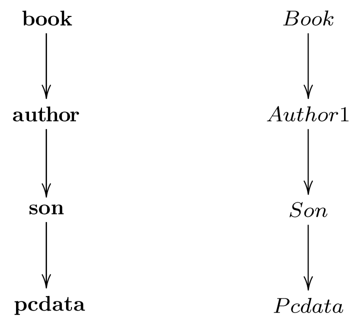
LEMMA 2.2. *Some regular tree languages are not single-type tree languages, and some single-type tree languages are not local tree languages.*

To prove this lemma, we only have to show that (1) the regular tree grammar $G_{2.1}$ in Example 2.1 cannot be captured by any single-type tree grammar and that (2) the single-type tree grammar $G_{2.3}$ in Example 2.3 cannot be captured by any local tree grammar. Both (1) and (2) can be easily proved by contradiction, and are thus omitted.

The next theorem directly follows from the above lemmas.

THEOREM 2.1. *The class of regular tree languages properly includes the class of single-type tree languages which in turn properly includes the class of local tree languages.*

Table I summarizes which class each of example grammars so far belongs to.




(a) Tree (b) Interpretation

Fig. 2. A tree and its interpretation against $G_{2.6}$.

## 2.5 Uniqueness of Interpretations

In this section, we consider whether or not a tree can have more than one interpretation against a grammar.

PROPOSITION 1. *Any tree has at most one interpretation against a local tree grammar.*

When a tree is valid against a local tree grammar, we can easily construct an interpretation of this tree as follows. For each node in this tree, we find a production rule having the terminal of this node in the right-hand side. Since we prohibited competition of nonterminals in local tree grammars, it is guaranteed that we find at most one such production rule. We determine that the nonterminal in the left-hand side of this production rule be the nonterminal for this node. It is obvious that this tree has no other interpretations.

*Example* 2.9. Consider the grammar $G_{2.6}$ in Example 2.6 and a valid tree:

```
<book><author><son>text chunk</son></author></book>
```

The nonterminals for the `book`, `author`, and `son` elements are *Book*, *Author*, and *Son*, respectively, and the nonterminal for the text node is *Pcdata*. Figure 2 depicts the tree and its unique interpretation.

Observe that the previous procedure for constructing interpretations does not require full validation. In fact, it does not even examine content models.

PROPOSITION 2. *Any tree has at most one interpretation against a single-type tree grammar.*

Given a single-type tree grammar and a tree valid against it, we can construct the interpretation of this tree as before. The only difference is that we have to resolve competition of nonterminals by using content models of parent nodes. As for the root node, it is guaranteed that competition does not occur. As for each nonroot node, we can assume that we have already determined the nonterminal for its parent node. Note that we can uniquely determine the content model for the parent node since no two production rules share the same nonterminal in the left-hand side and share the same terminal in the right-hand side at the same time. Since two different nonterminals in this content model do not compete, we can uniquely determine the nonterminal for the current node. As previously, it is obvious that this tree has no other interpretations.

*Example* 2.10. Consider the single-type tree grammar $G_{2.3}$ in Example 2.3 and a valid tree:

```
<book><author><son>text chunk</son></author></book>
```

Obviously, the nonterminal for the root `book` element is *Book*. Although nonterminals *Author1* and *Author2* compete for the terminal **author**, only the former appears in the content model for the nonterminal *Book*. Thus, the nonterminal for the `author` element is *Author1*. The nonterminal for the `son` element is obviously *Son*. Again, Figure 2 (used in the Example 2.9) depicts the tree and interpretation.

This procedure does not require full validation. Although it examines content models for resolving competition, it does not ensure if content models are satisfied or not. Unlike local or single-type tree grammars, regular tree grammars do not guarantee uniqueness of interpretations.

PROPOSITION 3. *A tree may have more than one interpretation against a regular tree grammar.*

We demonstrate a regular tree grammar that has multiple interpretations.

*Example* 2.11. The following regular tree grammar $G_{2.11} = (N, T, S, P)$ is not single-type. Observe that competing nonterminals *Para1* and *Para2* occur in the content model (*Para1**, *Para2**).

$$
\begin{aligned}
N &= \{Doc, Para1, Para2, Pcdata\} \\
T &= \{\mathbf{doc}, \mathbf{para}, \mathbf{pcdata}\} \\
S &= \{Doc\} \\
P &= \{Doc \rightarrow \mathbf{doc}\ (Para1^*, Para2^*), Para1 \rightarrow \mathbf{para}\ (Pcdata), \\
&\quad Para2 \rightarrow \mathbf{para}\ (Pcdata), Pcdata \rightarrow \mathbf{pcdata}\ (\epsilon)\}
\end{aligned}
$$

Uniqueness of interpretation does not hold for this grammar since *Para1* and *Para2* are interchangeable for the document `<doc><para/></doc>`.

This regular tree grammar is artificial since the distinction between *Para1* and *Para2* is unnecessary. If we merge these nonterminals, we obtain an equivalent single-type tree grammar which certainly ensures uniqueness of interpretations. However, as we have seen in the previous section, it is not always possible to rewrite a regular tree grammar to an equivalent single-type tree grammar.

As we will see in Section 5, uniqueness of interpretations is one of the most contentious issues surrounding XML schema languages. Some people consider uniqueness of interpretations to be crucial since they believe that validators *should pass* interpretations to application programs. Others do not care about uniqueness of interpretations, since they believe that validators *must not pass* interpretations to application programs. We will provide a concise overview of this controversy in Section 5.

## 2.6 Boolean Closure

The last topic in this section is boolean closure. In contrast to uniqueness of interpretations, boolean closure holds for regular tree languages but does not hold for single-type or local tree languages.

*Definition* 2.7. A class of languages is said to be closed under union (respectively, intersection and set difference), when, for any two languages in that class, their union (respectively, intersection and set difference) also belongs to the same class.

THEOREM 2.2. *The class of single-type tree languages and that of local tree languages are not closed under union.*

We can make a stronger claim: the union of two local tree languages is not always single-type. Consider two local tree grammars $G_{2.2.1}$ and $G_{2.2.2}$ as.[3]

$$\begin{aligned} G_{2.2.1} &= (\{Doc, Sec1, Para\}, \{\mathbf{doc}, \mathbf{sec}, \mathbf{para}\}, \{Doc\}, \\ &\qquad \{Doc \rightarrow \mathbf{doc}\ (Sec1^*), Sec1 \rightarrow \mathbf{sec}\ (Para), Para \rightarrow \mathbf{para}\ (\epsilon)\}), \\ G_{2.2.2} &= (\{Doc, Sec2, Para\}, \{\mathbf{doc}, \mathbf{sec}, \mathbf{para}\}, \{Doc\}, \\ &\qquad \{Doc \rightarrow \mathbf{doc}\ (Sec2^*), Sec2 \rightarrow \mathbf{sec}\ (Para, Para^+), Para \rightarrow \mathbf{para}\ (\epsilon)\}). \end{aligned}$$

The union of $L(G_{2.2.1})$ and $L(G_{2.2.2})$ can be captured by a regular tree grammar, but cannot be captured by a single-type tree grammar. In fact, it can be captured by a *regular* tree grammar $G_{2.2.3}$ shown here:

$$\begin{aligned} G_{2.2.3} &= (\{Doc, Sec1, Sec2, Para\}, \{\mathbf{doc}, \mathbf{sec}, \mathbf{para}\}, \{Doc\}, \\ &\qquad \{Doc \rightarrow \mathbf{doc}\ (Sec1^* \mid Sec2^*), \\ &\qquad Sec1 \rightarrow \mathbf{sec}\ (Para), Sec2 \rightarrow \mathbf{sec}\ (Para, Para^+), Para \rightarrow \mathbf{para}\ (\epsilon)\}. \end{aligned}$$

Observe that $G_{2.2.3}$ is not a single-type tree grammar, since nonterminals *Sec1* and *Sec2* compete with each other and occur in the content model of the first production rule. If we improve this grammar by merging *Sec1* and *Sec2*, we will allow unnecessary trees that are not valid against either $G_{2.2.1}$ or $G_{2.2.2}$. By contradiction, we can easily show that no single-type tree grammars capture the union.

THEOREM 2.3. *The class of single-type tree languages and that of local tree languages are not closed under set difference.*

Again, we can make a stronger claim: the set difference of two local tree languages is not always single-type. Consider two local tree grammars $G_{2.3.1}$ and $G_{2.3.2}$:

$$\begin{aligned} G_{2.3.1} &= (\{Doc, Sec1, Para\}, \{\mathbf{doc}, \mathbf{sec}, \mathbf{para}\}, \{Doc\}, \\ &\qquad \{Doc \rightarrow \mathbf{doc}\ (Sec1, Sec1), Sec1 \rightarrow \mathbf{sec}\ (Para^*), Para \rightarrow \mathbf{para}\ (\epsilon)\}), \end{aligned}$$

[3]For convenience, we use $A^+$ to denote $A, A^*$.

Table II. Summary of Closure Properties

| Grammar Class | Boolean Operation | | |
|---|---|---|---|
| | Union | Difference | Intersection |
| local tree grammar | Not Closed | Not Closed | Closed |
| single-type tree grammar | Not Closed | Not Closed | Closed |
| regular tree grammar | Closed | Closed | Closed |

$$G_{2.3.2} = (\{Doc, Sec2, Para\}, \{\mathbf{doc}, \mathbf{sec}, \mathbf{para}\}, \{Doc\}, \{Doc \rightarrow \mathbf{doc}\ (Sec2, Sec2), Sec2 \rightarrow \mathbf{sec}\ (Para^{+}), Para \rightarrow \mathbf{para}\ (\epsilon)\}).$$

The set difference $L(G_{2.3.1}) - L(G_{2.3.2})$ cannot be captured by a single-type tree grammar. In fact, it can be captured by a *regular* tree grammar $G_{2.3.3}$ defined as:

$$G_{2.3.3} = (\{Doc, Sec1, Sec2, Para\}, \{\mathbf{doc}, \mathbf{sec}, \mathbf{para}\}, \{Doc\}, \{Doc \rightarrow \mathbf{doc}\ ((Sec1, Sec2) \mid (Sec2, Sec1)), Sec1 \rightarrow \mathbf{sec}\ (Para^{*}), Sec2 \rightarrow \mathbf{sec}\ (\epsilon), Para \rightarrow \mathbf{para}\ (\epsilon)\}).$$

Observe that $G_{2.3.3}$ is not a single-type tree grammar since nonterminals *Sec1* and *Sec2* compete with each other and occur in the content model of the first production rule. By contradiction, we can easily show that no single-type tree grammars capture this set difference.

THEOREM 2.4. *The class of single-type tree languages and that of local tree languages are closed under intersection.*

We defer a formal proof to the Appendix, but give an informal overview. First, we construct an intersection grammar, say $G_3$, from two given grammars, say $G_1$ and $G_2$. For each nonterminal $X_1$ in $G_1$ and nonterminal $X_2$ in $G_2$, we introduce a nonterminal, denoted $X[X_1, X_2]$, for $G_3$. For each terminal **a**, we select $X_1 \rightarrow \mathbf{a}\ r_1$ from $G_1$ and $X_2 \rightarrow \mathbf{a}\ r_2$ from $G_2$. From $r_1$ and $r_2$, we create a regular expression $r_3$ over $N_1 \times N_2$ such that $r_3$ simulates both $r_1$ and $r_2$. We create a production rule $X[X_1, X_2] \rightarrow \mathbf{a}\ r_3$ for $G_3$. Then, we only have to show that this grammar is local or single-type when both $G_1$ and $G_2$ are local or single-type, respectively.

THEOREM 2.5. *The class of regular tree languages is closed under union, intersection, and set difference* [Takahashi 1975].

This result is well known and we do not provide a proof in this article. Interested readers are referred to Comon et al. [1997] and Takahashi [1975]. Closure properties under union, intersection, and difference are summarized in Table II.

## 3. XML SCHEMA LANGUAGES

In this section, using the three grammar classes that we introduced in Section 2, we study various representative XML schema language proposals: DTD, W3C XML Schema, RELAX NG. Our focus is on the mathematical properties of these schema languages in our framework.

We capture all these schema proposals by regular tree grammars. For this purpose, we slightly modify our definition of production rules. We allow production rules without terminals, that is, they are of the form $x \to r$, where $x \in N$ and $r$ is a regular expression over $N$. However, we impose a restriction that all such production rules can be safely expanded to regular expressions over nonterminals whose production rules have a terminal symbol on the right hand side.[4] For example, $x \to ((y, x, y)|z)$ is disallowed since this production rule causes non-regular string languages. Note that, given such a regular tree grammar, we can rewrite the grammar into one where all the production rules have a terminal in the right hand side as in Definition 2.1. After such rewriting, our definitions of interpretation and valid documents from Section 2 still hold. The notion of complex types and interpretations as defined by W3C XML Schema require further attention and are discussed in Section 3.2.

### 3.1 DTD

DTD as defined in Bray et al. [2004] is a local tree grammar. This is enforced by not distinguishing between terminals and nonterminals. Element type declarations of DTDs are production rules, and element types of XML 1.0 are terminals as well as nonterminals. Content models are required to be deterministic (see Section 6.6). Attribute-list declarations of DTDs associate attributes to terminals.

As an example, consider a DTD as follows:

```
<!ELEMENT doc (para*)>
<!ELEMENT para (#PCDATA)>
```

It can be captured by a local tree grammar:

$$N = \{Doc, Para, Pcdata\}$$
$$T = \{\mathbf{doc}, \mathbf{para}, \mathbf{pcdata}\}$$
$$S = \{Doc\}$$
$$P = \{Doc \to \mathbf{doc}\ (Para^*), Para \to \mathbf{para}\ (Pcdata), Pcdata \to \mathbf{pcdata}\ (\epsilon)\}$$

Weak expressive power of local tree grammars can be problematic in designing an XML schema using DTD. As an example, consider elements representing paper titles and elements representing section titles. DTD authors often would like to use different contents for these two types of title elements. However, if they use the same tag name `title` for both types, they are forced to write a single content model. As a result, they have to introduce many tag names such as `paperTitle`, `sectionTitle`, `subSectionTitle`, and so forth.

### 3.2 W3C XML Schema

The expressiveness of W3C XML Schema [Thompson et al. 2001] is mostly within that of the single-type grammars as intended by the specification. However, in some cases, it fails to be in single-type (see Section 3.2.7). As in DTDs,

[4]Fine details of this restriction are schema-language dependent.

content models in W3C XML Schema are required to be deterministic (see Section 6.6).

The main features of W3C XML Schema are complex type, anonymous type, model groups, derivation by extension and restriction, substitution groups, abstract type definitions, and integrity constraints such as key, unique and keyref constraints. Except for integrity constraints, the rest of the these features can be described in our framework.

3.2.1 *Complex Types.* A complex type defines a production rule without terminals. For instance:

```
<xsd:complexType name="Book">
  <xsd:sequence>
    <xsd:element name="title" type="xsd:string"
                 minOccurs="1 maxOccurs="1"/>
    <xsd:element name="author" type="xsd:string"
                 minOccurs="1" maxOccurs="unbounded"/>
    <xsd:element name="publisher" type="xsd:string"
                 minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
```

This can be converted into production rules *Book* $\rightarrow$ (*Title*, *Author*$^{+}$, *Publisher*$^{?}$), *Title* $\rightarrow$ **title** (*Pcdata*), *Author* $\rightarrow$ **author** (*Pcdata*), *Publisher* $\rightarrow$ **publisher** (*Pcdata*). The first production rule does not have terminals in the right-hand side. The other rules are required for specifying permissible values of child elements.

Element declarations refer to complex types with the attribute `type`. For example, `<xsd:element name="MyFavoriteBook" type="Book"/>` specifies that the content of `MyFavoriteBook` elements are of the type *Book*.

Interpretations as defined by W3C XML Schema require that we know the complex type associated with an element as well. In other words, $I(e)$ as defined in Definition 2.2 needs to include the complex type corresponding to the contents of $e$. For example, consider the following two element declarations in W3C XML Schema, `<xsd:element name="MyFavoriteBook" type="Book"/>` and `<xsd:element name="MyFavoriteBook" type="Book1"/>`. Here *Book* and *Book1* are complex types, with production rules, *Book* $\rightarrow$ (*RE*), *Book1* $\rightarrow$ (*RE1*). Now when we rewrite the above W3C XML Schema to a regular tree grammar as in Definition 2.1, we will get *MyFavoriteBookAsBook* $\rightarrow$ **MyFavoriteBook**(*RE*), and *MyFavoriteBookAsBook1* $\rightarrow$ **MyFavoriteBook**(*RE1*). Now after interpretation, we will know the complex types associated with the elements as well.

3.2.2 *Anonymous Complex Types.* Anonymous complex types are mapped to our framework by introducing new nonterminals and production rules. For example, consider the anonymous complex type for `item`, which is the second `xsd:complexType` in this example.

```
<xsd:complexType name="Items">
  <xsd:sequence>
```

```
    <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="productName" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

This can be converted by introducing a new nonterminal *Item* and the production rules *Items* $\rightarrow$ (*Item*$^*$), *Item* $\rightarrow$ **item**(*ProductName*), and *ProductName* $\rightarrow$ **productName** (*Pcdata*).

3.2.3 *Model Groups.* A model group definition defines a nonterminal and a production rule without a terminal. An example in Fallside [2001] is shown here:

```
<xsd:group name="ShipAndBill">
  <xsd:sequence>
     <xsd:element name="shipTo" type="USAddress"/>
     <xsd:element name="billTo" type="USAddress"/>
  </xsd:sequence>
</xsd:group>
```

This model group definition is equivalent to production rules: *ShipAndBill* $\rightarrow$ (*ShipTo*, *BillTo*), *ShipTo* $\rightarrow$ **shipTo** (*USAddress*), and *BillTo* $\rightarrow$ **billTo** (*USAddress*).

Model groups can be freely referenced from other model groups or complex types, for example, by `<xs:group ref="ShipAndBill"/>`.

3.2.4 *Derivation.* Complex types cannot be freely referenced from model groups or other complex types. *Derivation* is the only mechanism for defining complex types based on other complex types. Derivation is done by extension or restriction. An example of derived types by extension is given in the following. This example is borrowed from Fallside [2001] Section 4.1, but is modified slightly.

```
<xsd:complexType name="Address">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="UKAddress">
  <xsd:complexContent>
    <xsd:extension base="Address">
      <xsd:sequence>
```

```
        <xsd:element name="postcode" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

The complex type *UKAddress* is derived from the complex type *Address* by adding *Postcode*. These complex types are captured by production rules $Address \rightarrow (Name, Street, City)$, $UKAddress \rightarrow (Name, Street, City, Postcode)$, and production rules for *Name, Street, City, Postcode*.

Derivation by restriction creates a new complex type by imposing restrictions on another complex type. However, we are forced to write the whole content model rather than specifying the restrictions. In the following example, complex type *WashingtonAddress* is derived from *Address* by restriction but `name`, `street`, and `city` are specified again.

```
<xsd:complexType name="WashingtonAddress">
  <xsd:complexContent>
    <xsd:restriction base="Address">
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="street" type="xsd:string"/>
        <xsd:element name="city">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="Washington"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

The restriction imposed by complex type *WashingtonAddress* is that `city` must have `Washington` as the value. Validators complain that a complex type derived by restriction allows what is not allowed by the original. For example, validators raise an error if *WashingtonAddress* introduces a child element `postcode`.

W3C XML Schema provides an attribute called *final* which prevents derived types by extension or restriction. For example, if *Address* had defined *final* = *#all*, then we cannot derive *UKAddress* or *WashingtonAddress* from it.

3.2.5 `xsi:type`. An element declared to be of a complex type $A$ can actually be of another complex type $B$, if $B$ is derived from $A$ and the element specifies `xsi:type="B"`. As an example, recall the complex types *Address* and *UKAddress* introduced earlier. Although an element declaration `<xsd:element name="shipTo" type="Address"/>` references to *Address*, we can use *UKAddress* as the complex type of `shipTo` elements thus allowing `postcode`. To do so, we only have to specify `xsi:type="UKAddress"`.

```
<shipTo xsi:type="UKAddress">
      <name>Helen Zoe</name>
      <street>47 Eden Street</street>
      <city>Cambridge</city>
      <postcode>CB1 1JR</postcode>
</shipTo>
```

In our framework, `xsi:type` can be captured by introducing additional terminal symbols and production rules before validation. For example, `<xsd:element name="shipTo" type="Address"/>` is represented by *ShipTo* $\rightarrow$ **shipTo** (*Address*). We introduce another rule: *ShipTo* $\rightarrow$ **shipTo** `@xsi:type="UKAddress"`(*UKAddress*), where **shipTo**`@xsi:type="UKAddress"` is a "terminal symbol". This production rule can be applied to those `shipTo` elements having `xsi:type="UKAddress"`.

Such use of `xsi:type` can be prevented by specifying an attribute called *block* in the original type. For example, if *Address* had defined "*block=#all*", then `shipTo` elements cannot specify `xsi:type="UKAddress"`. Our framework can easily capture *block* by not introducing additional production rules as a side effect.

3.2.6 *Substitution Groups.* A substitution group definition allows some terminal symbols to be substituted by other terminal symbols. For example, consider an element declaration

```
<xsd:element name="comment" type="X"/>
```

This gets translated into the production rule *comment* $\rightarrow$ **comment** ($X$). Consider the substitution group definition (Fallside [2001] Section 4.6; we modify it slightly for easy explanation):

```
<xsd:element name="shipComment" type="Y"
         substitutionGroup="comment"/>
<xsd:element name="customerComment" type="Z"
         substitutionGroup="comment"/>
```

This substitution group specifies that the terminal **comment** can be replaced by the terminals **shipComment** or **customerComment**. W3C XML Schema requires that $Y$ and $Z$ are derived from $X$. This is converted into grammar rules:

$$\begin{aligned}
&\mathit{ShipComment} \rightarrow \textbf{shipComment}\ (Y),\\
&\mathit{CustomerComment} \rightarrow \textbf{customerComment}\ (Z),\\
&\mathit{comment} \rightarrow \textbf{shipComment}\ (Y), \text{ and}\\
&\mathit{comment} \rightarrow \textbf{customerComment}\ (Z)
\end{aligned}$$

3.2.7 *Wildcards.* Wildcards allow elements or attributes without specifying tag names. Wildcards sometimes lead to non-single-type schemas, however.

For example, the following schema[5] is not single-type.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="test">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:any namespace="##any" processContents="strict"/>
       <xsd:element name="foo" type="xsd:integer"/>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
 <xsd:element name="foo" type="xsd:string"/>
</xsd:schema>
```

In the content model for `test` elements, both the wildcard (`<xsd:any ...>`) and `<xsd:element name="foo" type="xsd:integer"/>` allow a terminal symbol `foo`. Note that the latter specifies integers as contents. The following XML document is valid against this schema although the content of the first `foo` element is not an integer.[6]

```
<test>
  <foo>bar</foo>
  <foo>1</foo>
</test>
```

The preceeding schema is rather a restrained-competition tree grammar which we will cover in Section 6.3.

3.2.8 *Miscellaneous.* W3C XML Schema has many more additional features. Abstract type definitions are similar to complex type definitions with the additional constraint that an abstract type should never occur in the instance document, rather its subtype should be used. Similarly, an element declared as abstract should not occur in the instance document, instead another element belonging to its substitution group should be used. These features can be captured in the framework of regular tree grammars. W3C XML Schema also supports specification of integrity constraints such as **keys, keyref** and **unique**. They specify additional constraints for an XML document to be valid against a schema. These constraints are similar to the integrity constraints present in the relational model such as primary key, foreign key, and unique key constraints. Such integrity constraints in W3C XML Schema cannot be captured using our framework but rather require an additional layer on top of it.

### 3.3 RELAX NG

RELAX NG can represent any regular tree grammar as did its predecessors RELAX Core [ISO/IEC 2001] and TREX [Clark 2001b]. RELAX NG represents production rules by define elements. The attribute `name` of a `define` element

[5]We owe this example to Eric van der Vlist.

[6]We do not translate wildcards into our grammar production rules, rather we require that the set of terminal symbols be finite. However we can easily extend our framework to capture wildcards by extending our alphabet to be an infinite set but with a finite set of equivalence classes.

specifies a nonterminal in the left-hand side. The child elements of the `define` element captures the right-hand side. A terminal symbol and a content model in the right-hand side are represented by a child `element` element and the children of this `element` element. Unlike the DTD and W3C XML Schema, RELAX NG allows nondeterministic content models (see Section 6.6).

To increase readability, RELAX NG allows production rules not to have a terminal in the right-hand side. Such production rules provide syntax sugar and can be safely expanded without loss of information. For example, consider the following RELAX NG schema:

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <ref name="AddressBook"/>
  </start>
  <define name="AddressBook">
    <element name="addressBook">
      <zeroOrMore>
        <ref name="Card"/>
      </zeroOrMore>
    </element>
  </define>
  <define name="Card">
    <element name="card">
      <ref name="Inline"/>
  </element>
  </define>
  <define name="Inline">
    <ref name="Name"/>
    <ref name="Email"/>
  </define>
  <define name="Name">
    <element name="name">
      <text/>
    </element>
  </define>
  <define name="Email">
    <element name="email">
      <text/>
    </element>
  </define>
</grammar>
```

Here *AddressBook* is a nonterminal that produces a tree, and *Inline* is a nonterminal that produces a list of trees. The above RELAX NG grammar will be represented in our framework as follows:

$$\begin{aligned} P = \{ & \mathit{AddressBook} \rightarrow \textbf{addressBook}\,(\mathit{Card}^*),\ \mathit{Card} \rightarrow \textbf{card}(\mathit{Inline}),\\ & \mathit{Inline} \rightarrow (\mathit{Name, Email}),\\ & \mathit{Name} \rightarrow \textbf{name}(\mathit{Pcdata}),\ \mathit{Email} \rightarrow \textbf{email}(\mathit{Pcdata}),\ \mathit{Pcdata} \rightarrow \textbf{pcdata}\,(\epsilon)\} \end{aligned}$$

RELAX NG has two significant extensions of regular tree grammars: attribute-element constraints and interleaving. We will cover these topics in Section 7.3.

## 4. DOCUMENT VALIDATION ALGORITHMS

In this section, we consider algorithms for document validation and describe time and memory requirements.

### 4.1 Preparations

In preparation, we introduce element automata. An element automaton is a usual string automaton; however, the alphabet is a set of nonterminals. We can create an element automaton from a content model $r$ by applying some well-known algorithm for constructing string automata from regular expressions [Hopcroft and Ullman 1979]. We denote the constructed element automaton as $M[r]$; $M[r]$ is represented by a 5-tuple $\{Q, \Sigma, \delta, q_0, Q_F\}$, where $\Sigma$ is a finite *alphabet*, $Q$ is a finite set of *states*, $q_0\,(\in Q)$ is a *start state*, $Q_F\,(\subseteq Q)$ is the set of *final states*, and $\delta$ is a function from $\Sigma \times Q$ to the power set of $Q$. Note that this definition allows nondeterminism, since the transition function returns a *set* of states. By executing this element automaton, we can determine whether or not a given sequence of nonterminals matches the content model. We illustrate the execution of element automata below for completeness.

Given a sequence $X_1 X_2 \ldots X_n$ of nonterminals, we execute an element automaton $M$ by applying $\delta$ repeatedly. That is, we compute sets of states

$$\begin{aligned} Q_0 &= \{q_0\}, \\ Q_1 &= \{q \mid q \in \delta(X_1, q'), q' \in Q_0\}, \\ Q_2 &= \{q \mid q \in \delta(X_2, q'), q' \in Q_1\}, \ldots \\ Q_n &= \{q \mid q \in \delta(X_n, q'), q' \in Q_{n-1}\}. \end{aligned}$$

If some state in $Q_n$ is a final state in $Q_F$, then $X_1 X_2 \ldots X_n$ is accepted by $M$. Otherwise, it is not accepted.

### 4.2 Validation Against Local Tree Grammars

Remember that a tree has at most one interpretation against a local or single-type tree grammar. This uniqueness allows simple algorithms for validation against local or single-type tree grammars.

We begin with validation for local tree grammars which do not allow competing nonterminals (see Definition 2.5). We validate an XML document while traversing it in a depth-first manner. When we visit an element $e$, we can uniquely determine a nonterminal $n$ and a content model $r$ from this start tag. When we exit from $e$, we examine if the nonterminal sequence assigned to the child elements of $e$ matches $r$ by executing $M[r]$ against this nonterminal sequence. This idea is effected by Algorithm 1. It uses a stack $S$ of nonterminal lists. Each nonterminal list $l$ contains nonterminals assigned to sibling elements. This list is created as an empty list when a start tag is encountered. A nonterminal is appended to this list when a child element is left.

**Algorithm 1:** `Validation for local tree grammars`

**Input** : XML document D

Let $S$ be an empty stack of lists of nonterminals;
Let $Y$ be an empty stack of production rules;
**traverse** $D$ **in the depth-first manner**
  **when** *element* $e$ **is visited**
    find a production rule $X \rightarrow \mathbf{a}\ (r)$ such that $\mathbf{a}$ is the tag name of $e$;
    //At most one such production rule is found.
    **if** *no rule is found* **then**
      report "invalid" and halt;
    push $X \rightarrow \mathbf{a}\ (r)$ to $Y$;
    push an empty list to $S$;
  **when** *element* $e$ **is exited from**
    pop $X \rightarrow \mathbf{a}\ (r)$ out of $Y$;
    pop a list $(X_1, X_2, \ldots, X_n)$ out of $S$;
    //$X_1, X_2, \ldots, X_n$ are nonterminals assigned to the children of $e$;
    **if** $M[r]$ *does not accept* $(X_1, X_2, \ldots, X_n)$ **then**
      report "invalid" and halt;
    append $X$ to the nonterminal list at the top of $S$;
report "valid";

We have to extend Algorithm 1 for handling text nodes. Ideally, we only have to handle a text node as an element. That is, when we encounter a text node, we perform the action for visiting it and that for leaving from it. However, XML has a design flaw: whitespace is used for tag indentation and is also allowed as part of the document content. Thus, we have to discard whitespace used for indentation and handle other text nodes as elements. This part is tricky and is beyond the scope of this article.

Observe that Algorithm 1 not only determines whether a document is valid but also constructs a unique interpretation of the document. If we skip execution of element automata, this algorithm constructs an interpretation without full validation.

We can improve Algorithm 1 by executing $M[r]$ step-by-step. That is, whenever we determine $X_i$, we compute a state set $Q_i$ as defined in Section 4.1. This improvement allows early detection of invalid documents: when $Q_i$ becomes empty, we can immediately report that the document is invalid. However, to keep our algorithms simple, we have not incorporated this improvement.

### 4.3 Validation Against Single-Type Tree Grammars

Now, we extend Algorithm 1 for handling single-type tree grammars. This extension is quite simple since single-type tree grammars also ensure uniqueness of interpretations.

Remember that a single-type tree grammar does not allow competing start symbols and does not allow competing nonterminals within a single content model (see Definition 2.6). Thus, for each element, we can uniquely determine a nonterminal. This idea is effected by Algorithm 2. It uses another

**Algorithm 2:** `Validation for single-type tree grammars.`

**Input** : XML document $D$
Let $S$ be an empty stack of lists of nonterminals;
Let $Y$ be an empty stack of production rules;
Let $P$ be an empty stack of nonterminal sets;
Push the set of start symbols into $P$;
**traverse** $D$ **in the depth-first manner**
  **when** *element* $e$ **is visited**
    find a production rule $X \rightarrow \mathbf{a}\ (r)$ such that $\mathbf{a}$ is the tag name of $e$ and $X$ is contained in the nonterminal set at the top of $P$;
    //At most one such production rule is found.
    **if** *no such* $X$ *is found* **then**
      report "invalid" and halt;
    push $X \rightarrow \mathbf{a}\ (r)$ to $Y$;
    push an empty list to $S$;
    push the set of nonterminals occurring in $r$ into $P$;
  **when** *element* $e$ **is exited from**
    pop $X \rightarrow \mathbf{a}\ (r)$ out of $Y$;
    pop a list $(X_1, X_2, \ldots, X_n)$ out of $S$;
    $//X_1, X_2, \ldots, X_n$ are nonterminals assigned to the children of $e$;
    **if** $M[r]$ *does not accept* $(X_1, X_2, \ldots, X_n)$ **then**
      report "invalid" and halt;
    append $X$ to the nonterminal list at the top of $S$;
    pop a nonterminal set out of $P$;
report "valid";

stack $P$ for maintaining the set of permissible nonterminals for the current element. If the current element is the root element, this set is the set of start symbols. Otherwise, it is the set of nonterminal sets occurring in the content model for the parent element. Even when more than one production rule is found for the current element, at most one of these production rule has its left-hand-side nonterminal in the nonterminal set at the top of the stack.

Our observations on Algorithm 1 apply to Algorithm 2. That is, it constructs a unique interpretation of a valid document, and we can improve this algorithm by executing element automata step-by-step.

### 4.4 Validation Against Regular Tree Grammars

Remember that a tree may have more than one interpretation against a regular tree grammar. This nonuniqueness complicates validation. Unlike Algorithms 1 and 2, our algorithm for handling regular tree grammars cannot choose one nonterminal when they encounter a start tag, but rather have to keep track of multiple candidates at the same time.

In preparation, we reconsider element automata. We have executed an element automaton against a sequence of nonterminals. But it is possible to execute an element automaton against a sequence of *sets* of nonterminals.

**Algorithm 3:** Validation for `regular tree grammars`.

**Input** : XML document $D$
Let $S$ be an empty stack of lists of sets of nonterminals;
//Note that we have to use *sets* of nonterminals rather than nonterminals.
Let $Y$ be an empty stack of *sets* of production rules;
//Note that we have to use *sets* of production rules rather than
//production rules.
Let $P$ be an empty stack of nonterminal sets;
Push the set of start symbols into $P$;
**traverse** $D$ **in the depth-first manner**
  **when** *element* $e$ **is visited**
    find production rules of the form $X^i \to \mathbf{a}\ (r^i)$ such that $\mathbf{a}$ is the tag name of $e$ and $X^i$ is contained in the nonterminal set at the top of $P$;
    //More than one such production rule may be found.
    //$X^i$ is an applicable nonterminal.
    **if** *no such production rule is found* **then**
      report "invalid" and halt;
    push $\{X^i \to \mathbf{a}\ (r^i) \mid i = 1, 2, \ldots\}$ to $Y$;
    push an empty list to $S$;
    push the set of nonterminals occurring in some $r^i$ into $P$;
  **when** *element* $e$ **is exited from**
    pop $\{X^i \to \mathbf{a}\ (r^i) \mid i = 1, 2, \ldots\}$ out of $Y$;
    pop a list of sets of nonterminals $(\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n)$ out of $S$;
    //$\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n$ are *sets* of nonterminals assigned to
    //the children of $e$;
    let $\mathbf{X}$ be the set of $X^i$ such that $M[r^i]$ accepts $(\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n)$;
    **if** $\mathbf{X}$ *is empty* **then**
      report "invalid" and halt;
    append $\mathbf{X}$ to the list of sets of nonterminals at the top of $S$;
    pop a nonterminal set out of $P$;
report "valid";

Let $\mathbf{X}_1\mathbf{X}_2\ldots\mathbf{X}_n$ be a sequence of sets of nonterminals. We execute an element automaton $M$ by applying $\delta$ repeatedly for every element in $\mathbf{X}_i (1 \le i \le n)$. That is, we compute sets of states

$$
\begin{aligned}
Q_0 &= \{q_0\},\\
Q_1 &= \{q \mid q \in \delta(X_1, q'), q' \in Q_0, X_1 \in \mathbf{X}_1\},\\
Q_2 &= \{q \mid q \in \delta(X_2, q'), q' \in Q_1, X_2 \in \mathbf{X}_2\}, \ldots\\
Q_n &= \{q \mid q \in \delta(X_n, q'), q' \in Q_{n-1}, X_n \in \mathbf{X}_n\}.
\end{aligned}
$$

If some state in $Q_n$ is a final state in $Q_F$, then $\mathbf{X}_1\mathbf{X}_2\ldots\mathbf{X}_n$ is accepted by $M$. Otherwise, it is not accepted. $\mathbf{X}_1\mathbf{X}_2\ldots\mathbf{X}_n$ is accepted if and only if we can choose some nonterminal $X_i$ from $\mathbf{X}_i$ for every $i$ and the nonterminal sequence $X_1X_2\ldots X_n$ is accepted by $M$. Observe that some nonterminals in $\mathbf{X}_i$ may be useless (i.e., never chosen) even when $\mathbf{X}_1\mathbf{X}_2\ldots\mathbf{X}_n$ is accepted. For example, $X_n \in \mathbf{X}_n$ is useless when $\{q \mid q \in \delta(X_n, q'), q' \in Q_{n-1}\}$ and $Q_F$ are disjoint.

Now, we are ready to introduce Algorithm 3 , which is an extension of Algorithm 2. The main differences are that (1) more than one production rule may be found for each element, and (2) a set of nonterminals (rather than a single nonterminal) is assigned to each element. Because of (1), we use $X^i$, $r^i$, and $M[r^i](i = 1, 2, \ldots)$ rather than $X$, $r$, and $M[r]$. Because of (2), we use lists of *sets* of nonterminals rather than lists of nonterminals. Note that $\mathbf{X}_j$ $(1 \leq j \leq n)$ is a set of nonterminals. If the element automaton $M[r^i]$ accepts the sequence of nonterminal sets $(\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n)$, then $X^i$ is added to $\mathbf{X}$.

As in Algorithms 1 and 2, we can improve this algorithm by executing element automata step-by-step. This improvement does complicate this algorithm but allows early detection of invalid documents.

Unlike Algorithms 1 and 2, this algorithm does not construct a unique interpretation of the document and does not enumerate all interpretations (see Section 5.4 for further discussion). This algorithm enumerates all applicable nonterminals for each element when the validator exits from it, but some of them may be found useless when the validator exits from its parent. To illustrate, suppose that a document `<doc><sec/><sec><para/></sec></doc>` is validated against $G_{2.3.3}$ in Theorem 2.3. When the validator exits from the empty `<sec/>` element, it enumerates both *Sec1* and *Sec2* as applicable nonterminals, but *Sec1* is found useless when it exits from the `<doc>` element.

### 4.5 Tree Model vs. Event Model

Programs for handling XML documents, including validators, are typically implemented on top of APIs for XML such as DOM [Wood et al. 1998] and SAX [Megginson 2000]. These APIs are based on either the tree or event model.

In the *tree model*, the XML parser reads an entire XML document and creates a tree in memory. Then, via tree-model APIs (e.g., DOM), application programs have access to the tree in memory and traverse it any number of times. A drawback of the tree model is that a significantly large amount of memory is required when the XML document is very large.

On the other hand, in the *event model*, the XML parser does not create a tree in memory, but rather raises events when it encounters start or end tags. Then an application program is notified of the events by event-model APIs (e.g., SAX) and takes appropriate actions. As another way to implement the event model, pull APIs such as Stax [Fry 2003] have recently appeared. While SAX push events to application programs, pull APIs allow application programs to explicitly request (pull) events from the XML parser. It has been argued that pull APIs are more programmer-friendly than push APIs.

Note that all of the Algorithms 1, 2, and 3 can be implemented on top of both tree and event-model APIs (both push and pull APIs).

### 4.6 Complexity

Now, let us consider the complexity of the three aforementioned Algorithms. In our experiences, the space required by documents is far more significant than space required by validation algorithms. In other words, the distinction between the event and tree models is more important than space complexity. Therefore, we focus on the time complexity, ignoring the space complexity. Especially, we study complexity with respect to the size of documents rather than the size of schemas.

Our algorithms (Algorithms 1, 2, and 3) examine each start or end tag only once through depth-first-search scan. For each start or end tag, these algorithms perform some action. The document size does not affect the time required by this action. Thus, the time complexity of these algorithms is linear to the size of documents.

## 5. DISCUSSIONS

In this section, we consider which class of tree grammars is appropriate as a basis for XML schema languages. It is generally agreed that local tree grammars are less appropriate as a basis for XML schema languages since the expressive power is significantly weaker than the other classes. However, it is controversial whether regular tree grammars are more appropriate than single-type grammars or vice versa.

### 5.1 Expressiveness

The expressive power of local tree grammars is weak as we observed in Section 2.4. In the history of SGML and XML, this weakness has hindered the use of XML for representing narrative documents. As an example, consider elements representing paper titles and elements representing section titles. DTD authors often would like to use slightly different contents for these two kinds of title elements. However, if they use the same tag name `title` for both kinds, they are forced to write a single content model. As a result, they have to introduce many tag names such as `paperTitle`, `sectionTitle`, `subSectionTitle`, and so forth. The proliferation of such tag names hinders document editing, programming, DTD maintenance, querying, and so forth.

Local tree grammars also hinder the use of XML for representing data. Although programmers naturally expect local scoping for data representations, local scoping is blocked by local tree grammars. For example, if `<x>` subelements of `<Point>` are integers, then `<x>` subelements of `<Foo>` in the same document are forced to be integers. Furthermore, local tree grammars fail to capture syntactic constraints imposed by HTML (Similar problems arise in the design of other schemas such as DocBook [Walsh and Muellner 1999].)

—Anchor (`<a>`) elements of HTML are not allowed to nest, even indirectly. For example, `<a>...<span>...<a>...</a>...</span> ... </a>` is prohibited, although `<a>` elements may contain `<span>` elements and vice versa. Likewise, form (`<form>`) elements of HTML are not allowed to nest, even indirectly. For example, `<form>...<div>...<form>...</form>...</div> ...</form>` is prohibited, although `<div>` elements may contain `<form>` elements and vice versa.

—Paragraph (`<p>`) elements of HTML 2.0 can contain input (`<input>`) elements only when the `p` elements are descendants of some `form` elements. For example, `<form>...<p>...<input>...</input>...</p>...</form>` is allowed, although `<body><p>...<input>...</input>...</p></body>` is prohibited.

As a remedy to the weakness of local tree grammars, SGML [ISO 8879 1986] allows *inclusion* and *exclusion exceptions* to accompany content models. An exclusion exception *disallows* the occurrence of an element even when it is *allowed* by the content model. Likewise, an inclusion exception *allows* the occurrence of an element even when it is *not allowed* by the content model. Inclusion and exclusion exceptions control not only children but also descendants. For example, if the exclusion of anchor (a) elements is specified at the element type declaration of anchor elements, then `<p><a>...<span><a>...</a></span> ... </a></p>` is disallowed, even when the content model for span allows anchor elements.

The introduction of inclusion and exclusion exceptions allows SGML DTDs to capture nonlocal (but regular) tree languages. The SGML DTDs for HTML 2 [Berners-Lee and Connolly 1995], 3.2 [Raggett 1997], and 4.01 [Raggett et al. 1999] uses inclusion or exclusion exceptions for representing restrictions as shown. In particular, the SGML DTD for HTML 2.0 uses the combination of inclusion and exclusion for capturing the constraint that every input element must have a form element as an ancestor. However, as more element types are introduced, this approach quickly became too complicated. HTML 3.2 and 4.0 dropped this constraint, thus allowing <input> without ancestor <form>.

XML did not inherit inclusion or exclusion exceptions from SGML. Although the W3C XML WG was aware of their advantages, the WG concluded that they are too complicated for implementors and schema authors. As a result, the DTD of XML is restricted to local tree languages.

Both single-type or regular tree grammars are free from the problems shown earlier. For example, (1) the use of nonterminals such as *PaperTitle* and *SectionTitle* allows <title> elements to have different content models, and (2) the use of nonterminals *LineStart* and *TrainStart* allow <start> subelements of <Line> to be integers and allow those of <Train> to be time.

Constraints required by HTML can also be captured by single-type or regular tree grammars. However, such a single-type or regular tree grammar is quite lengthy. For example, to allow or disallow <a>, <em>, <span>, <strong>, etc. depending on ancestor elements, we have to introduce nonterminals *aWithAWithEMWithSPAN, aWithAWithEMWithoutSPAN, aWithAWithoutEMWithSPAN, aWithAWithoutEMWithoutSPAN*, and so forth. Some syntactic sugar is required for making schemas more compact and easier to understand.

The expressive power of single-type tree grammars is weaker than that of regular tree grammars as we observed in Section 2.4. In other words, some schemas (e.g., $G_{2.1}$) can not be captured by single-type tree grammars but can be captured by regular tree languages. At present, no industrial schemas require the expressiveness power of regular tree grammars. But we are not sure if this is because there are no such requirements or if this is because of the lack of sufficiently expressive schema languages With the advent of RELAX NG, such schemas may appear in the future.

## 5.2 Validation

Our validation algorithm for local tree grammars, namely Algorithm 1, is simpler than our algorithms for single-type or regular tree grammars. Existing validators for DTDs use a variation of Algorithm 1. Specifically, Xerces2 Java[7] (an XML parser in Java) uses a variation of Algorithm 1 on the basis of the event model.

Single-type grammars require Algorithm 2, which is more complicated than Algorithm 1. "Schema-validity assessment", as defined in 7.2 of Thompson et al. [2001], is similar to Algorithm 2. To the best of our knowledge, all implementations of W3C XML Schema follow this reference model. For example, on the basis of the event model, Xerces2 Java uses a variation of Algorithm 2. However, schema-validity assessment differs from Algorithm 2 in that it uses the current state of an element automaton for uniquely determining the nonterminal for the current element. This difference is due to the fact that W3C XML Schema deviates from single-type tree grammars to restrained-competition tree grammars (see Section 6.3).

Regular tree grammars require more advanced algorithms such as Algorithm 3. Two validators[8] for RELAX Core use Algorithm 3 on the basis of the event model. However, Algorithm 3 is not powerful enough for TREX and RELAX NG since they are equipped with attribute-element constraints and interleaving. We will consider validation algorithms for RELAX NG in Section 7.

### 5.3 Boolean Closure

We have considered three operations: union, intersection, difference. The class of local tree languages and that of single-type tree languages is closed under intersection but is not closed under union or difference. However, the class of regular tree languages is closed under each of these operations.

Type inference for XML programming or query languages [Hosoya and Pierce 2003; Christensen et al. 2003; Tozawa 2001; Milo et al. 2000; Murata 1997] are based on these operations. The intersection operation is typically used for the type inference of pattern matching. That is, the intersection of the tree language representing a pattern and that representing a schema becomes the type of the pattern matching expression. The difference operation can be used for checking subtyping or detecting if a tree language $L_1$ is a subset of another tree language $L_2$. That is, $L_1$ is a subtype of $L_2$ exactly when the set difference between $L_1$ and $L_2$ is empty.[9]

### 5.4 Interpretations

Uniqueness of interpretations is strongly related with a fierce controversy about XML. This controversy is called "XML class warfare" between bohemians and gentry [Ogbuji 2002].

This warfare stems from a basic difference of opinions on the use of schemas. One camp (gentry) believes that XML documents should be annotated with type information obtained from schemas. This camp further believes that

[7]Available at http://xml.apache.org/xerces2-j/index.html.

[8]They are RELAX verifier for Java and RELAX verifier for C++. For about them, see http://www.xml.gr.jp/relax/.

[9]Two efficient algorithms (Hosoya and Pierce [2002] and Tozawa and Hagiya [2003]) do not use the difference operation for subtyping, however.

validators should provide interpretations (type information) to application programs. Uniqueness of interpretations is considered crucial under this scenario. Supporters of W3C XML Schema typically belong to this camp. Meanwhile, the other camp (bohemians) supports RELAX NG. Bohemians believe that XML documents without type information are the heart of XML, and that validators must not provide interpretations to application programs.[10] Under this scenario, uniqueness of interpretations is not necessary. It is worth noting that bohemians do not oppose the use of schemas as types. They merely want to introduce another layer, namely, data binding tools (e.g., JAXB [Fordin 2003] and Relaxer[11]) or XML-aware programming languages (e.g., XDuce [Hosoya and Pierce 2003] and JWIG [Christensen et al. 2003]), for such use of schemas. Bohemians certainly want to keep the basic layer of XML free from types. It is interesting that data binding tools or XML-aware programming languages may or may not require uniqueness of interpretations (e.g., XDuce does not).

We do not examine arguments of the two camps further since we emphasize a formal approach but this controversy is rather a software architecture issue. Interested readers are referred to Ogbuji [2002] and van der Vlist [2002].

Finally, recall that Algorithms 1 and 2 construct unique interpretations, but Algorithm 3 does not. The second camp (bohemians) sees no problems in Algorithm 3 (and other algorithms in Section 7), while the first camp (gentry) does.

## 6. OTHER GRAMMATICAL CONCEPTS

In this section, we discuss other grammatical concepts which are closely related with tree grammars. They are context-free grammars, balanced context-free grammars, restrained-competition tree grammars, regular hedge grammars, and deterministic content models.

### 6.1 Context-Free Grammars

Some readers might wonder why we do not use context-free grammars. Context-free grammars [Hopcroft and Ullman 1979] represent sets of strings. Successful parsing of strings against context-free grammars provides derivation trees. This scenario is appropriate for programming languages and natural languages where programs and natural language text are strings.

However, start and end tags in an XML document directly represent a tree. The XML parser reconstructs this tree without using a schema. The XML validator then receives this tree as an input and validates it against a schema. Thus, tree grammars are much more appropriate for representing schemas than context-free grammars are.

Nevertheless, early works on document schemas used context-free grammars as schemas. XML documents are not represented by sentences of such context-free grammars. Rather, they are represented by derivation trees.

[10]Bohemians do not even allow schemas to specify default values. Indeed, the RELAX NG specification does not provide default values.
[11]Available at http://relaxer.org/.

*Example* 6.1. $G_{2.6}$ in Example 2.6 can be represented as a context-free grammar as shown.

$$\begin{aligned} N &= \{Book, Author1, Son, Pcdata\} \\ T &= \{\} \\ S &= \{Book\} \\ P &= \{Book \rightarrow (Author1), Author1 \rightarrow (Son), \\ &\quad Son \rightarrow (Pcdata), Pcdata \rightarrow (\epsilon)\} \end{aligned}$$

Note that the right-hand side of a production rule has a regular expression of nonterminals but does not have terminals. In fact, this context-free grammar has no terminal symbols and allows no strings except the null string.

One can assume that such use of context-free grammars mimic local tree grammars. In fact, it is straightforward to create such context-free grammars from local tree grammars and vice versa. However, this approach cannot capture W3C XML Schema and RELAX NG since they require the expressive power of single-type or regular tree grammars.

### 6.2 Balanced Context-Free Grammars

Notwithstanding the limitations shown in the previous section, it is possible to use context-free grammars to capture W3C XML Schema and RELAX NG. The key idea is to represent trees as strings by using start-and end-parenthesis symbols. For example, a tree `<doc><para/><para>text chunk</para></doc>` can be represented as a string comprising seven symbols: `<doc>`, `<para>`, `</para>`, `<para>`, `text chunk`, `</para>`, and `</doc>`, where `<doc>` and `<para>` are start-parenthesis symbol and `</doc>` and `</para>` are end-parenthesis symbol. This representation allows us to use context-free grammars for describing XML documents as sentences.

*Example* 6.2. The regular tree grammar $G_{2.1}$ shown in Example 2.1 can be captured as a context-free grammar.

$$\begin{aligned} N &= \{Doc, Para1, Para2, Pcdata\} \\ T &= \{\texttt{<doc>}, \texttt{</doc>}, \texttt{<para>}, \texttt{</para>}, \textbf{pcdata}\} \\ S &= \{Doc\} \\ P &= \{Doc \rightarrow \texttt{<doc>}Para1, Para2^{*}\texttt{</doc>}, \\ &\quad Para1 \rightarrow \texttt{<para>}\texttt{</para>}, Para2 \rightarrow \texttt{<para>}Pcdata\texttt{</para>}, \\ &\quad Pcdata \rightarrow \textbf{pcdata}\} \end{aligned}$$

In such context-free grammars, the right-hand side of each production rule has a regular expression of nonterminals surrounded by a start-parenthesis and end-parenthesis pair, where **pcdata**-only production rules are exceptions. Such specialized context-free grammars, called balanced context-free grammars, are studied by Berstel and Boasson [2002] and Brüggemann-Klein and Wood [2004]. Balanced context-free grammars and regular tree grammars are equally expressive. Although no validation algorithms are presented in Berstel and Boasson [2002], balanced context-free grammars help with understanding the derivative-based validation shown in Section 7.2.

### 6.3 Restrained-Competition

We have considered three classes of regular tree grammars. However, another class called restrained-competition deserves some attention. This class allows competition of nonterminals, but requires that it is restrained by content models.

*Definition* 6.1. A content model $r$ restrains competition of two competing nonterminals $A$ and $B$ if, for any sequences $U, V, W$ of nonterminals, either $U\,A\,V$ or $U\,B\,W$ fails to match $r$.

*Definition* 6.2. A restrained-competition tree grammar is a regular tree grammar such that

—for each production rule, its content model restrains competition of nonterminals occurring in the content model, and
—start symbols do not compete with each other.

*Example* 6.3. Nonterminals *Para1* and *Para2* in the grammar $G_{2.1}$ compete with each other, and they both occur in the content model of the production rule for *Doc*. However, this content model (*Para1*, *Para2*$^*$) restrains the competition between *Para1* and *Para2*, since *Para1* may occur only as the first nonterminal and *Para2* may occur only as the nonfirst nonterminal. Thus, $G_{2.1}$ is a restrained-competition tree grammar.

*Example* 6.4. The grammar $G_{2.11}$ is not a restrained-competition tree grammar. Observe that the content model (*Para1*$^*$, *Para2*$^*$) does not restrain the competition of nonterminals *Para1* and *Para2*. For example, suppose that $U = V = W = \epsilon$. Then, both $U$ *Para1* $V$ and $U$ *Para2* $W$ match this content model.

A set of trees is a *restrained-competition tree language* if, for some restrained-competition tree grammar, all trees in this set are valid and no other trees are valid. It is not hard to show that the class of restrained-competition tree languages properly contains the class single-type and is properly contained in the class regular.

We can easily extend Algorithm 2 for handling restrained-competition tree grammars. Recall that, when a start tag is encountered, the algorithm finds a production rule from the tag name and the content model for the parent element. To handle restrained-competition tree grammars, we only have to take the nonterminals assigned to elder sibling elements into consideration.

One could argue that this class has some advantages: it is more expressive than the class single-type, while ensuring uniqueness of interpretations and allowing a simple validation algorithm.

### 6.4 Regular Hedge Grammars

Although we consider trees and tree grammars in this article, we can extend our framework for handling hedges. A *hedge* is a sequence of zero or more trees. Regular hedge grammars differ from regular tree grammars in two points: (1) the start symbol of a regular hedge grammar is a regular expression comprising pairs of nonterminals and terminals (a regular expression over $N \times T$), and (2) production rules of a regular hedge grammar are of the form $X \rightarrow r$ such that $r$ is a regular expression over $N \times T$.

*Example* 6.5. If we reformulate $G_{2.1}$ as a regular hedge grammar, the start symbol is a pair ($\mathbf{doc}, Doc$) and the production rules are:

$$P = \{Doc \rightarrow (\mathbf{para}[Para1], \mathbf{para}[Para2]*), \\ Para1 \rightarrow \epsilon,\ Para2 \rightarrow \mathbf{pcdata}[Pcdata], Pcdata \rightarrow \epsilon\}$$

It is easy to convert regular tree grammars to regular hedge grammars. But the converse is not always possible since hedges are not always trees. Researchers (e.g., Takahashi [1975]) found regular hedge grammars more naturally extend regular string grammars than regular tree grammars .

### 6.5 Validation by Tree Automata

Just as regular grammars for strings can be recognized by automata, regular tree grammars can be recognized by tree automata. Validation of trees against regular tree grammars can be considered as execution of tree automata.

Tree automata have been extensively studied [Comon et al. 1997]. A tree automaton examines a given tree by assigning states to nodes in the tree. The tree automaton accepts the tree if it terminates at one of the final states.

There are top-down tree automata and bottom-up tree automata: the former begins with the root node and assigns states to elements after handling superior nodes, while the latter begins with leaf nodes and assigns states to nodes after handling subordinate nodes. Moreover, there are deterministic tree automata and nondeterministic tree automata: the former assigns a state to each node, while the latter assigns any number of states to each node. As a result, there are four types of tree automata: deterministic top-down, nondeterministic top-down, deterministic bottom-up, and nondeterministic bottom-up.

It is known that nondeterministic top-down, deterministic bottom-up and nondeterministic bottom-up tree automata are equally expressive [Comon et al. 1997]. In other words, any regular tree language is accepted by some nondeterministic top-down automaton. Likewise, it is also accepted by some deterministic bottom-up automaton and accepted by some nondeterministic bottom-up tree automaton. Meanwhile, deterministic top-down tree automata are not equally expressive. In other words, some regular tree languages *cannot* be accepted by any deterministic top-down tree automata.

Algorithms 1 and 2 are similar to deterministic top-down automata. However, deterministic top-down tree automata assign a state to a node without examining that node; they only examine the parent node and the state assigned to it. Because of this restriction, deterministic top-down tree automata are almost useless for XML. On the other hand, both Algorithms 1 and 2 examine an element before assigning a nonterminal (state) to it.

Algorithm 3 can be seen as a combination of nondeterministic top-down and nondeterministic bottom-up as follows; (1) nondeterministic top-down—when this algorithm visits an element, it computes a set of nonterminal candidates, and (2) nondeterministic bottom-up—when this algorithm leaves an element, it chooses some of these nonterminal candidates.

It is possible to use deterministic bottom-up tree automata for validation. Given a regular tree grammar, we create a deterministic bottom-up tree automaton. This is done by introducing a state for each subset of the set of nonterminals of the grammar and then constructing a transition function for these states. Execution of this deterministic bottom-up tree automaton is straightforward which is the biggest advantage of this approach. However, a drawback of this approach is that subset construction may cause combinatorial explosion.

### 6.6 Deterministic Content Models

DTDs and W3C XML Schema impose the constraint that content models be deterministic. This constraint is called *Unique Particle Attribution* in W3C XML Schema and is called *one-unambiguous* by Brüggemann-Klein and Wood [1998].

A content model is deterministic if, during pattern matching, we can always choose one symbol in the content model. Formally, a content model is deterministic when a Glushkov automaton [Glushkov 1961] constructed from it is already deterministic [Brüggemann-Klein and Wood 1998].

*Example* 6.6. `(a,b) | (a,c)` is a nondeterministic content model. The reason is that we cannot choose one of the two occurrences of `a` when we encounter `a` in a sequence `ac`. Only after examining `c` in this sequence, can we choose the second occurrence of `a`. Meanwhile, an equivalent content model `(a,(b | c))` is deterministic since it has only one occurrence of `a`.

*Example* 6.7. `(a*,a)` is a nondeterministic content model. When we encounter the first `a` in a sequence `aa`, we cannot choose one of the two occurrences of `a` in this content model. Only after examining the entire sequence, can we choose the first occurrence of `a`. Meanwhile, an equivalent content model `(a,a*)` is deterministic since we can choose the first occurrence of `a` when we encounter the first `a` in a given sequence, and we can only choose the second occurrence of `a` for any other `a`.

*Example* 6.8. `((a,b)*,a?)` is a nondeterministic content model since we cannot choose one of the two occurrences of `a` when we encounter `a` in a sequence `aba`. (Note that this regular expression allows both `a` and `b` as the last symbol.) No deterministic content models exactly capture this content model.

Deterministic content models were first introduced by SGML and have been thoroughly studied by Brüggemann-Klein and Wood [1998]. They have shown that deterministic content models cannot capture some regular languages and further shown that the union of some deterministic content models cannot be captured by any deterministic content model. For example, the union of `(a,(b,a)*)` and `(a,b)*`, both of which are deterministic, is equivalent to the content model in Example 6.8.

There has been considerable of debate about deterministic content models. Deterministic content models make schema authoring difficult, and they break boolean closure. The proponents of deterministic content models, however, argue that deterministic content models make implementations easier and faster.

It is important to note that none of the algorithms described in Section 4 require that content models be deterministic. In particular, our validation algorithm (Algorithm 2) for single-type tree grammars works for nondeterministic content models. In other words, single-type tree grammars and deterministic content models are orthogonal issues. One can design a schema language restricted to single-type tree grammars with or without deterministic content models.

## 7. MORE SOPHISTICATED ALGORITHMS FOR RELAX NG VALIDATION

Although validators for regular tree grammars can be built using Algorithm 3, modern validators for RELAX NG use more sophisticated algorithms. This section sketches these algorithms, namely, binarization and derivative-based validation, briefly.

There are two advantages to these algorithms. First, validation becomes simpler and more efficient since they do not construct an element automaton from each content model. (In our experiences, this construction deteriorates the performance of validators.) Second, they can be easily extended for handling attribute-element constraints and the interleaving of content models, both of which RELAX NG is equipped with.

### 7.1 Binarization

Our trees and tree grammars allow a node to have any number of children. However, it is possible to convert trees and tree grammars to binary trees and binary tree grammars, respectively. Any nonleaf node in a binary tree has exactly two children. This binarization makes validation simpler and more efficient since it becomes unnecessary to create an element automaton from each content model. Two validators for RELAX NG (Bali[12] and Miaou[13]) are based on binarization.

It is well known that a tree can be represented by a binary tree [Knuth 1973]. A node in the binary tree represents either a node in the original tree or a null node, denoted $\epsilon$. Hereafter, we use "bt-node" to mean nodes in binary trees. The left child bt-node and right child bt-node of a bt-node represent the eldest child node and the immediately following sibling node in the original tree, respectively. As a special case, the null bt-node as the left child implies that the original node have no children, and the null bt-node as the right child implies that the original node have no younger siblings. For example, a binary tree representing `<sec><sec><p/></sec><sec><p/></sec><sec/></sec>` is depicted in Figure 3.

[12]Available at http://www.kohsuke.org/relaxng/bali/doc/.

[13]Available at http://www.idealliance.org/papers/xml02/dx_xml02/papers/06-00-14/06-00-14.html.

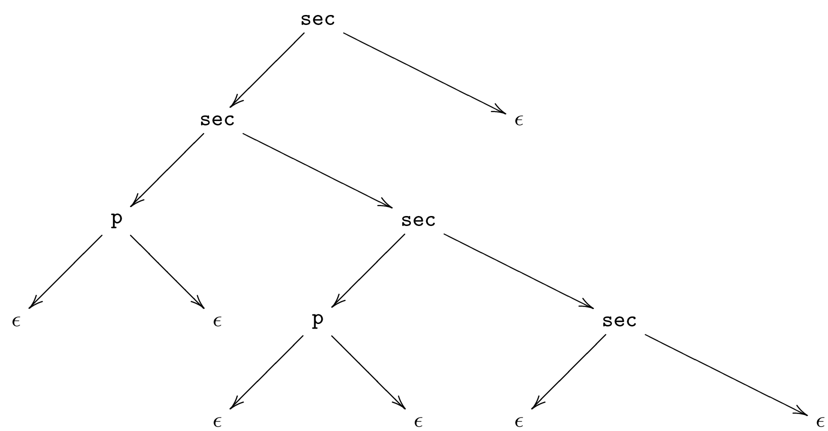



Fig. 3. A binary tree representing <sec><sec><p/></sec><sec><p/></sec><sec/></sec>.

On the basis of this representation, it is possible to convert a regular tree grammar to a binary tree grammar [Takahashi 1975]. A binary tree grammar [Comon et al. 1997] represents a set of binary trees. Hosoya et al. [2003] (see Appendix A in their article) presented an efficient algorithm for converting tree grammars to binary tree grammars. That algorithm does not construct element automata from content models.

*Example* 7.1. To illustrate, we use a regular tree grammar:

$$\begin{aligned} N &= \{\mathit{Root}, \mathit{Sec}, P\} \\ T &= \{\mathbf{sec}, \mathbf{p}\} \\ S &= \{\mathit{Root}\} \\ P &= \{\mathit{Root} \to \mathbf{sec}\ (\mathit{Sec}^*), \mathit{Sec} \to \mathbf{sec}\ ((\mathit{Sec}|P)^*), P \to \mathbf{p}\ (\epsilon)\} \end{aligned}$$

From this grammar, we can construct a binary tree grammar shown as:

$$\begin{aligned} N &= \{\mathit{Final}, \mathit{\_Root}, \mathit{\_Sec}, \mathit{Emp}\} \\ T &= \{\mathbf{sec}, \mathbf{p}\} \\ S &= \{\mathit{Final}\} \\ P &= \{\mathit{Final} \to \mathbf{sec}(\mathit{\_Root}, \mathit{Emp}),\ \mathit{\_Root} \to \mathbf{sec}(\mathit{\_Sec},\ \mathit{\_Root}),\ \mathit{\_Root} \to \epsilon, \\ &\quad \mathit{\_Sec} \to \mathbf{sec}(\mathit{\_Sec},\ \mathit{\_Sec}),\ \mathit{\_Sec} \to \mathbf{p}(\mathit{Emp}, \mathit{\_Sec}),\ \mathit{\_Sec} \to \epsilon, \mathit{Emp} \to \epsilon\} \end{aligned}$$

Note that the right-hand side of production rules of binary tree grammars are either of the form $\mathbf{a}\ (x_1, x_2)$ or $\epsilon$, where $\mathbf{a}$ is a terminal and $x_1, x_2$ are nonterminals.

As in Section 2, we can easily define interpretations and validity of binary trees against binary tree grammars. Figure 4 depicts an interpretation of the binary tree shown in Figure 3. Subtrees generated from nonterminals *_Root* and *_Sec* are permissible contents of first- and second-level sections, respectively.

Having introduced conversion of trees and regular tree grammars to binary trees and binary tree grammars, respectively, we can validate an XML document by validating a binary tree against a binary tree grammar.

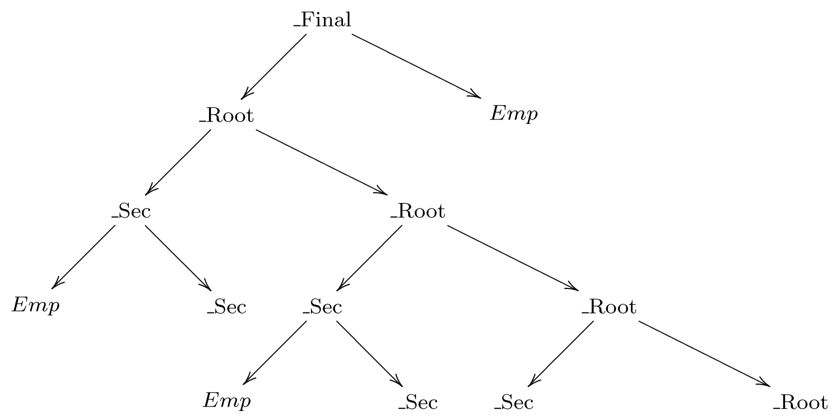



Fig. 4. An interpretation of the binary tree shown in Figure 3.

Algorithm 4 is a binary-tree version of Algorithm 3. While traversing a binary tree in depth-first, the preorder, in-order, and postorder actions are performed for each bt-node and an action is performed for each empty bt-node. The preorder action and postorder action are performed when the bt-node is visited and left, respectively. The in-order action is performed after the left child of the bt-node is left and before the right child is visited.

For each bt-node, this algorithm constructs two sets of nonterminals. The first set, denoted $C$, contains the candidate nonterminals. The second set, denoted $E$, is a subset of $C$ and contains the elected nonterminals. While $C$ is constructed before all descendant bt-nodes are examined, $E$ is constructed after all descendant bt-nodes have been examined. The initial value of $C$ is the set of start symbols (i.e., the candidates for the root node). The preorder and in-order actions for bt-node $e$ construct $C$ for the left and right child bt-nodes of $e$, respectively. The postorder action for $e$ constructs $E$. As a special case, $E$ is also constructed for null bt-node $\epsilon$.

For each bt-node, this algorithm also constructs three sets (namely $P$, $P'$, and $P''$) of production rules, where $P \supseteq P' \subseteq P''$. The preorder action computes $P$ from the tag name and $C$ for this bt-node. The in-order action computes $P'$ from $P$ as well as $E$ for the left child bt-node. The postorder action computes $P''$ from $P'$ as well as $E$ for the right child bt-node. A stack $Y$ of production rules maintains $P$, $P'$, and $P''$ for each bt-node.

Let us examine how this algorithm validates the binary tree in Figure 3 against the binary tree grammar in Example 7.1.

—The initial value of $C$ is a singleton set containing *_Final*.

—The preorder action for the first <sec> (the root bt-node) finds *Final* $\rightarrow$ **sec**(*_Root*, *Emp*), pushes a singleton set containing this rule to $Y$, and sets $\{$*_Root*$\}$ as the value of $C$.

—The preorder action for the second <sec> finds *_Root* $\rightarrow$ **sec**(*_Sec*, *_Root*), pushes a singleton set containing this rule to $Y$, and sets $\{$*_Sec*$\}$ as the value of $C$.

**Algorithm 4:** `Validation of binary trees`

**Input** : Binary tree D

Let $E$ be an empty set; //a set of elected candidates
Let $Y$ be an empty stack of *sets* of production rules;
Let $C$ be the set of start symbols; //a set of candidates
**traverse** $D$ **in the depth-first manner**
  **preorder for** *bt-node* $e$
    let $\mathbf{P}$ be the set of production rules of the form $X \to \mathbf{a}\ (X^{\mathrm{l}}, X^{\mathrm{r}})$ such that $\mathbf{a}$ is the tag name of $e$ and $X$ is contained in $C$;
    **if P** *is empty* **then**
      report "invalid" and halt;
    push $\mathbf{P}$ to $Y$;
    let $C$ be $\{X^{\mathrm{l}} \mid X \to \mathbf{a}\ (X^{\mathrm{l}}, X^{\mathrm{r}}) \in \mathbf{P}\}$;
  **in-order for** *bt-node* $e$
    pop $\mathbf{P}$ out of $Y$;
    let $\mathbf{P}'$ be $\{X \to \mathbf{a}\ (X^{\mathrm{l}}, X^{\mathrm{r}}) \in \mathbf{P} \mid X^{\mathrm{l}} \in E\}$;
    **if P′** *is empty* **then**
      report "invalid" and halt;
    push $\mathbf{P}'$ to $Y$;
    let $C$ be $\{X^{\mathrm{r}} \mid X \to \mathbf{a}\ (X^{\mathrm{l}}, X^{\mathrm{r}}) \in \mathbf{P}'\}$;
  **postorder for** *bt-node* $e$
    pop $\mathbf{P}'$ out of $Y$;
    let $\mathbf{P}''$ be $\{X \to \mathbf{a}\ (X^{\mathrm{l}}, X^{\mathrm{r}}) \in \mathbf{P}' \mid X^{\mathrm{r}} \in E\}$;
    **if P″** *is empty* **then**
      report "invalid" and halt;
    Let $E$ be $\{X \mid X \to \mathbf{a}\ (X^{\mathrm{l}}, X^{\mathrm{r}}) \in \mathbf{P}''\}$;
  **null bt-node** $\epsilon$
    let $\mathbf{P}$ be the set of production rules of the form $X \to \epsilon$ such that $X$ is contained in $C$;
    **if P** *is empty* **then**
      report "invalid" and halt;
    let $E$ be $\{X \mid X \to \epsilon \in \mathbf{P}\}$;
**if** $E$ *contains some start symbols* **then**
  report "valid";
**else**
  report "invalid";

—The preorder action for the first <p> finds *_Sec* → **p**(*Emp*, *_Sec*), pushes a singleton set containing this rule to $Y$, and sets {*Emp*} as the value of $C$.
—The action for the first null bt-node finds *Emp* → $\epsilon$ and sets {*Emp*} as the value of $E$.
—The in-order action for the first <p> pops a singleton set containing *_Sec* → **p**(*Emp*, *_Sec*) out of $Y$, pushes the same set back to $Y$, and sets {*_Sec*} as the value of $C$.
—The action for the second null bt-node finds *_Sec* → $\epsilon$ and sets {*_Sec*} as the value of $E$.

—The postorder action for the first <p> pops a singleton set containing $\_Sec \rightarrow \mathbf{p}(Emp, \_Sec)$ out of $Y$, and sets $\{\_Sec\}$ as the value of $E$.

—The in-order action for the second <sec> pops a singleton set containing $\_Root \rightarrow \mathbf{sec}(\_Sec, \ \_Root)$ out of $Y$, pushes the same set back to $Y$, and sets $\{\_Root\}$ as the value of $C$, and so forth.

This algorithm detects invalid documents as early as possible. This is because the in-order action uses elected nonterminals ($E$) of elder sibling elements for finding candidate nonterminals ($C$) of younger sibling elements. When no candidate nonterminals are found, the next preorder action fails to find production rules and thus reports an error. Meanwhile, Algorithm 3 may delay detecting invalid documents, since it does not execute the element automaton until all child elements are handled.

### 7.2 Derivative-Based Validation

James Clark designed a novel algorithm for validation against regular tree grammars in 2002,[14] and we describe the idea of derivatives-based validation algorithm briefly. For rigorous treatment, refer to Clark [2002].

His algorithm is based on *derivatives* of regular expressions [Brzozowski 1964]. The derivative of a regular expression $e$ with respect to a symbol x is a regular expression for what is left of $e$ after matching x. That is, it is a regular expression that matches any sequence that, when appended to x, will match $e$. For example, the derivative of a regular expression $ab$ with respect to a is $b$. Similarly, the derivatives of $ab \mid ac$ and $a*$ with respect to a are $b \mid c$ and $a*$, respectively.

The easiest way to understand this algorithm is to use balanced context-free grammars (as shown in Section 6.2). As in Section 6.2, we represent a document as a sequence of start- or end-parenthesis symbols.[15]

Our derivative expressions are regular expressions over terminal and nonterminal symbols. We begin with the right-hand side of the production rule for the start symbol. For each symbol in the sequence representing the document, we repeatedly construct a derivative expression. If we encounter a derivative expression that does not allow any sequence, we report that the document is invalid. After examining the entire sequence, we report that the document is valid if and only if the derivative expression allows the empty sequence (see Algorithm 5).

The rules for computing derivative expressions are given by the following equations. The first set of equations is borrowed from the standard definition of derivatives. Here ∅ is a regular expression representing the empty set.

$$\mathrm{deriv}(\epsilon, x) = \emptyset \tag{1}$$

$$\mathrm{deriv}(\emptyset, x) = \emptyset \tag{2}$$

$$\mathrm{deriv}(A \mid B, x) = \mathrm{deriv}(A, x) \mid \mathrm{deriv}(B, x) \tag{3}$$

[14] Jing (http://thaiopensource.com/relaxng/jing.html), a RELAX NG validator in Java, uses this algorithm.

[15] *After* patterns in Clark [2002] amount to end-parenthesis symbols of balanced context-free grammars.

**Algorithm 5:** `Derivative-based Validation`

**Input** : Document $D$ as sequence of start- and end-parenthesis symbols
Let $E$ be a regular expression; // the current derivative expression
Let $x$ be the current symbol; // from the document
Initialize $E$ to the right-hand side of the production rule for the start symbol;
**repeat**
  Get current symbol $x$ from $D$;
  Compute deriv$(E, x)$ and let $E$ be the result;
  **if** *$E$ does not allow any sequence* **then**
    report "invalid" and halt;
**until** *$D$ is empty*;
**if** *$E$ allows the empty sequence ($\epsilon$)* **then**
  // the derivative expression after reading all symbols allows $\epsilon$
  report "valid";
**else**
  report "invalid";

$$\text{deriv}(A^*, x) = \text{deriv}(A, x)A^* \tag{4}$$

$$\text{deriv}(AB, x) = \begin{cases} \text{deriv}(A, x)\,B & (A \text{ does not allow } \epsilon) \\ \text{deriv}(A, x)\,B \mid \text{deriv}(B, x) & (\text{otherwise}) \end{cases} \tag{5}$$

The second set of equations handles terminal symbols. Since we have start- or end-parenthesis symbols, we need two equations.

$$\text{deriv}(\texttt{<a>}, x) = \begin{cases} \epsilon & (x = \texttt{<a>}) \\ \emptyset & (\text{otherwise}) \end{cases} \tag{6}$$

$$\text{deriv}(\texttt{</a>}, x) = \begin{cases} \epsilon & (x = \texttt{</a>}) \\ \emptyset & (\text{otherwise}) \end{cases} \tag{7}$$

Finally, since nonterminals occur in content models, we have to dereference them. The right-hand side of the production rule for nonterminal $X$ is denoted RHS($X$).

$$\text{deriv}(X, x) = \text{deriv}(\text{RHS}(X), x) \tag{8}$$

As an example, consider the regular tree grammar shown in Example 7.1. A balanced context-free grammar equivalent to this regular tree grammar is:

$$\begin{aligned} N &= \{Root, Sec, P\} \\ T &= \{\texttt{<sec>},\ \texttt{</sec>},\ \texttt{<p>},\ \texttt{</p>}\} \\ S &= \{Root\} \\ P &= \{Root \to \texttt{<sec>}\ Sec^*\ \texttt{</sec>}, Sec \to \texttt{<sec>}\ (Sec|P)^*\ \texttt{</sec>}, \\ &\quad P \to \texttt{<p>}\ \epsilon\ \texttt{</p>}\} \end{aligned}$$

Let us validate an XML document `<sec><sec/><sec><p/></sec></sec>`. This document is represented by a sequence (`<sec>`, `<sec>`, `</sec>`, `<sec>`, `<p>`,

</p>, </sec>, </sec>). We begin with the right-hand side of the production rule for the start symbol, namely, <sec> $Sec^*$ </sec>.

The first symbol is <sec>. By (5) and (6), the first derivative expression is

$Sec^*$ </sec>

Let us construct the second derivative expression with respect to the second symbol <sec>. By (5), deriv($Sec^*$</sec>, <sec>) is deriv($Sec^*$, <sec>)</sec>| deriv(</sec>, <sec>). Since deriv(</sec>, <sec>) $= \emptyset$ by (7), the preceeding expression is equal to deriv($Sec^*$, <sec>)</sec>, which becomes deriv($Sec$, <sec>)$Sec^*$</sec> by (4). By applying (8) for $Sec$, we have deriv(<sec>$(Sec|P)^*$</sec>, <sec>) $Sec^*$ </sec>. Finally, by (5) and (6), we have

$(Sec|P)^*$</sec>$Sec^*$</sec>

The third derivative expression with respect to the next symbol </sec> is

$Sec^*$</sec>

Note that neither $Sec$ nor $P$ begin with </sec> and that $(Sec|P)^*$ allows the empty sequence.

The rest of this validation is done similarly. The following table shows the validation steps.

| next symbol | constructed derivative |
|---|---|
| | <sec> $Sec^*$ </sec> |
| The first <sec> | $Sec^*$ </sec> |
| The second <sec> | $(Sec|P)^*$ </sec> $Sec^*$ </sec> |
| The first </sec> | $Sec^*$ </sec> |
| The third <sec> | $(Sec|P)^*$ </sec> $Sec^*$ </sec> |
| <p> | </p> $(Sec|P)^*$ </sec> $Sec^*$ </sec> |
| </p> | $(Sec|P)^*$ </sec> $Sec^*$ </sec> |
| The second </sec> | $Sec^*$ </sec> |
| The third </sec> | $\epsilon$ |

This algorithm constructs derivatives lazily. In other words, it constructs a derivative only when it encounters the next symbol in the given document. This laziness ensures that the algorithm always terminates since the number of symbols in the given document is finite. Meanwhile, if we construct derivatives nonlazily (i.e., sufficiently to validate any document), we may end up with an infinite number of derivatives. For example, documents valid against the example grammar may have an arbitrary depth. Documents having third-level sections require a derivative

$(Sec|P)^*$</sec>$(Sec|P)^*$</sec>$Sec^*$</sec>

and those having fourth-level sections require a derivative

$(Sec|P)^*$</sec>$(Sec|P)^*$</sec>$(Sec|P)^*$</sec>$Sec^*$</sec>,

and so forth. A nonlazy procedure has to enumerate all such derivatives and thus will not terminate.

Although laziness ensures termination, it does not make this algorithm efficient. The optimization techniques such as Memoization (shown in Clark [2002]) makes this algorithm astoundingly efficient.

This algorithm has two particular advantages. First, validators become simpler since the in-memory representation is no longer automata but derivative expressions which are closer to the surface syntax of schema languages. Second, validation of small documents is remarkably efficient since lazy construction of derivative expressions does not touch unused content models.

### 7.3 Attribute-Element Constraints and Interleaving

RELAX NG provides two significant extensions of regular tree grammars: attribute-element constraints and interleaving. Both of them are inherited from TREX [Clark 2001b].

Although we have considered elements only, XML documents also contain attributes. If constraints on attributes can be obtained from terminals or nonterminals, validation of attributes is not difficult. DTD and W3C XML Schema provide only such constraints. However, RELAX NG can handle constraints between attributes and child elements. That is, we cannot determine permissible attributes of an element without examining its child elements. The expressive power yielded by this extension is quite substantial. For example, we can specify that a `person` element has either the attribute `name` or a subelement `name` but not both. Two validation algorithms for handling attribute-element constraints of RELAX NG have appeared. One is an extension of the derivative-based validation [Clark 2002] where we treat attributes as tokens and add rules for computing derivatives with respect to attribute tokens. The other is automaton rewriting [Hosoya and Murata 2002] where by examining attributes, attribute transitions in (binary) tree automata are removed or rewritten as null-transitions. The addition of attribute-element constraints preserves boolean closure [Hosoya and Murata 2003].

The interleaving of two regular languages is a language created by shuffling two sentences. For example, the interleaving of $A$ and $B$ is captured by $(A, B)|(B, A)$. The interleaving of $A^*$ and $B^*$ is captured by $(A|B)^*$. RELAX NG provides the interleave operator for combining patterns. This operator is typically used for representing mixed content models (e.g., the interleave of `<text/>` and $A$ provides `<mixed><ref name="A"/><text/>`) and for allowing any occurrence order (e.g., the interleave of $A$, $B$, and $C$ allows 6 possibilities). Two validation algorithms for handling interleaving have been presented. One (used for jing) [Clark 2001a] extends the derivative-based validation by computing the derivative of interleave expressions. The other (used for Bali) executes shuffle automata [Jedrzejowicz and Szepietowski 2001] which are equipped with branching and merging transitions.

## 8. RELATED WORK

To the best of our knowledge, it was Kil-Ho Shin [1992] who first used tree automata for structured documents. He used tree automata for the study of document formatting.

In the original XML working group, the XML Schema working group, and the XML Query Languages workshop, Murata (e.g., Murata [1999a, 1999b]; Murata and Robie [1999]) proposed that tree automata be used as the basis of schema and query languages. However, this proposal was not accepted by the XML Schema working group. Although we have seen an interpretation of W3C XML Schema using tree automata, it is an afterthought with some pitfalls. In a Japanese committee, Murata designed RELAX Core [Murata 2000] as an alternative schema language and submitted it to ISO/IEC. Around the same time, Klarlund et al. [1999] designed a schema language DSD which is also based on tree automata. Influenced by RELAX Core as well as XDuce [Hosoya and Pierce 2003] and XQuery [Boag et al. 2005], Jame Clark designed TREX [Clark 2001b]. Finally, RELAX NG [Clark and Murata 2001] was designed at OASIS by unifying RELAX Core and TREX and was standardized at ISO without any technical changes [ISO/IEC 2003].

Lee and Chu [2000] attempt to compare and classify more than ten schema languages (including W3C XML Schema and RELAX NG) from various perspectives. However, their approaches are by and large not mathematical so that the precise description and comparison among schema language proposals are not straightforward. On the other hand, this article and its precursor, Murata et al. [2001], first establish a formal framework based on regular tree grammars, and then compare schema language proposals.

XML Schema Formal Description of W3C (formerly called MSL [Brown et al. 2001]) is a mathematical model of W3C XML Schema. However, it is tailored for W3C XML Schema and is thus unable to capture other schema languages. Meanwhile, our framework is not tailored for a particular schema language. As a result, all schema languages can be captured.

Although we have considered DTD, W3C XML Schema, and RELAX NG only, other schema languages for XML documents are of considerable interest. Here we consider such languages.

First, there are schema languages proposed by researchers. Two notable examples are the type system of XDuce [Hosoya and Pierce 2003], and DSD (Document Structural Description) [Klarlund et al. 2000]. Both languages are based on regular tree languages.

The type system of XDuce is expressive enough to represent any regular tree language. XDuce is thus quite similar to RELAX NG. In fact, they have influenced each other. Differences between RELAX NG and XDuce are either syntactical ones or fine details such as mixed content and data types.

DSD 1.0 can represent any single-type tree grammars. DSD 1.0 can further represent some regular tree grammars which are not single-type. However, the designers of DSD 1.0 deliberately avoided the full power of regular tree grammars, but required rather that their top-down validation algorithm assigns at most one nonterminal to each element. As a result, DSD 1.0 is restricted to restrained-competition tree grammars (see Section 6.3). Thus, any XML document has at most one interpretation, and the union/intersection/difference of two DSD 1.0 schemas cannot always be constructed. DSD 2.0 is more expressive than DSD 1.0 in that it can represent any regular tree grammar. Thus, an XML document may have more than one interpretation, and the union/intersection/difference of two DSD 2.0 schemas can always be constructed.

Second, there are special purpose schema languages. Such a schema language is dedicated to a particular type of information which *may* be represented by XML. Primary constructs for such information are not elements or attributes. For example, RDF Schema [Brickley and Guha 2004] is dedicated to RDF metadata which consists of resources, properties, and statements, and the Topic Map Constraint Language[16] (under development at ISO/IEC) is dedicated to topic maps which consists of topics and associations. Special purpose schema languages are often more powerful than general purpose ones since they directly handle constructs specific to the problem domain.

Third, there are languages (e.g., Schematron [Jelliffe 2000]) for representing identity constraints. Identity constraints were originally developed for the relational database system, but they have been extended to XML by many researchers (e.g., Buneman et al. [2002]). RELAX NG does not provide any mechanisms for specifying identity constraints but was rather intended to interwork with identity-constraint languages. Meanwhile, W3C XML Schema is a stand-alone language equipped with identity constraint mechanisms. As of this writing, Schematron, is being standardized at ISO/IEC JTC1 SC34, as Part 3 of the Document Schema Definition Languages (ISO/IEC 19757).

Fourth, there is a family of languages (e.g., RELAX Namespace [Murata 2002] and NRL [Clark 2003]) for namespace-based validation dispatching. These languages allow the interworking of schemas describing different markup vocabularies and further allow these schemas to be written in different schema languages. For example, a RELAX NG schema for XHTML2 and a W3C XML Schema for XForms can be easily combined. A compound XML document is validated against this combination by dispatching elements in the XHTML2 namespace to a RELAX NG validator and dispatching elements in the XForms namespace to a W3C XML Schema validator. At the time of this writing, a latest member in this family, namely Namespace-based Validation Dispatching Language [Murata 2005], is being standardized at ISO/IEC JTC1 SC34, as Part 4 of the Document Schema Definition Languages (ISO/IEC 19757).

## 9. CONCLUSION

To compare XML schema language proposals, we have studied three classes of tree languages, namely, local, single-type, and regular. The class regular is the most expressive and is closed under boolean operations but does not ensure uniqueness of interpretations. The class single-type is less expressive, and the class local is even less expressive. These classes are not closed under union and difference operations but ensure uniqueness of interpretations. We have also presented validation algorithms for these classes. Those for local or single-type tree grammars are straightforward since they construct one interpretation per document. Those for regular tree grammars have to consider more than one

[16]The latest draft is available at http://www.isotopicmaps.org/tmcl/.

interpretation. We propose an algorithm for regular tree grammars which does not construct an interpretation but in stead, only reports whether the document is valid or not. Then, we have shown that DTD, W3C XML Schema, and RELAX NG are captured by local, single-type, and regular, respectively.

After introducing the three classes of tree languages, we study other grammatical concepts that are closely related to them. First, we studied how context-free grammars relate to tree grammars. Second, we introduced another class of tree grammars called restrained-competition which sits between single-type and regular. Third, we introduced regular hedge grammars. Fourth, we compared our validation algorithms and top-down/bottom-up deterministic/nondeterministic tree automata. Fifth, we introduced a restriction that content models be deterministic and studied how this restriction relates to our framework.

Finally, we introduced a validation algorithm based on binarization and another based on derivatives. They are more appropriate than our algorithms in Section 4 for implementing RELAX NG validators. One reason is that attribute-element constraints and interleaving of RELAX NG can be easily implemented on top of these algorithms.

Although schema languages and validators have been extensively studied, validator implementation is not a fully developed area. To the best of our knowledge, the number of fully-conformant validators for W3C XML Schema and RELAX NG is surprisingly small. We hope that the algorithms presented in this article provide a basis for future implementations.

## APPENDIX

In this appendix, we prove that the class of local tree languages and that of a single-type tree languages are closed under intersection.

In preparation, we show that the class of regular tree languages is closed under intersection. Let two regular tree grammars $G_1 = (N_1, T_1, P_1, S_1)$ and $G_2 = (N_2, T_2, P_2, S_2)$, respectively. Without loss of generality, we can assume that $T_1 = T_2 = T$. We construct a regular tree grammar that captures the intersection of $L(G_1)$ and $L(G_2)$.

Given regular expressions $r_1$ over $N_1$ and $r_2$ over $N_2$, we construct their *intersection* $r_1 \oplus r_2$, which is a regular expression over $N_1 \times N_2$. A sequence $(n_1^1, n_2^1)\,(n_1^2, n_2^2)\ldots(n_1^i, n_2^i)$ of nonterminals in $N_1 \times N_2$ matches $r_1 \oplus r_2$ exactly when $n_1^1 n_1^2 \ldots n_1^i$ matches $r_1$ and $n_2^1 n_2^2 \ldots n_2^i$ matches $r_2$.

This construction has four steps: (1) we create $r_1'$ (a regular expression over $N_1 \times N_2$) from $r_1$ by replacing each $n$ in $N_1$ by $(n_1, n_2^1)\,|\,(n_1, n_2^2)\,|\,\ldots\,|(n_1, n_2^k)$, where $n_2^1, n_2^2, \ldots, n_2^k$ is an enumeration of $N_2$. Similarly, we create $r_2'$ (another regular expression over $N_1 \times N_2$) from $r_2$; (2) we create two automaton from $r_1'$ and $r_2'$; (3) we create an intersection automaton of these automata; and (4) we create a regular expression, namely $r_1 \oplus r_2$, from this intersection automaton.

Now, we are ready to construct the intersection of $G_1$ and $G_2$. The intersection grammar is $G_3 = (N_1 \times N_2, T, P_3, S_1 \times S_2)$, where

$$P_3 = \{(n_1, n_2) \to \mathbf{a}\,(r_1 \oplus r_2)\,|\, n_1 \to \mathbf{a}\, r_1 \in P_1, n_2 \to \mathbf{a}\, r_2 \in P_2\}.$$

Obviously, a tree is valid against $G_3$ if and only if it is valid against $G_1$ as well as $G_2$.

It remains to show that (1) $G_3$ is local if $G_1$ and $G_2$ are local, and (2) $G_3$ is single-type if $G_1$ and $G_2$ are single-type. We prove (2) only, since (1) can be similarly proved.

First, we show that different start symbols in $S_1 \times S_2$ do not compete. Let two different start symbols be $(n_1^1, n_2^1)$ and $(n_1^2, n_2^2)$, where either $n_1^1 \neq n_1^2$ or $n_2^1 \neq n_2^2$. We consider the case that $n_1^1 \neq n_1^2$ only. By definition, $n_1^1$ and $n_1^2$ are start symbols of $G_1$. Since $G_1$ is single-type, $n_1^1$ does not compete with $n_1^2$; that is, if two production rules in $P_1$ have $n_1^1$ and $n_1^2$ respectively in the left-hand side, they do not share the same terminal symbol in the right-hand side. Consider two production rules in $P_3$ having $(n_1^1, n_2^1)$ and $(n_1^2, n_2^2)$. They do not share the same terminal symbol in the right-hand side, since they are created from production rules having $n_1^1$ and $n_1^2$ in the left-hand side. Therefore, $(n_1^1, n_2^1)$ and $(n_1^2, n_2^2)$ do not compete.

Second, we show that different nonterminal symbols occurring in a single content model do not compete. Consider a content model $r^3$ in $G_3$. By definition, $r^3 = r^1 \oplus r^2$ for some content model $r^1$ in $G_1$ and $r^2$ in $G_2$. Let two different nonterminals occurring in $r^3$ be $(n_1^1, n_2^1)$ and $(n_1^2, n_2^2)$, where either $n_1^1 \neq n_1^2$ or $n_2^1 \neq n_2^2$. Again, we consider the case that $n_1^1 \neq n_1^2$ only. Both $n_1^1$ and $n_1^2$ occur in $r^1$, because $r^3$ is created from $r^1$ (and $r^2$). Since $G_1$ is single-type, $n_1^1$ and $n_1^2$ do not compete. As previously, we can show that $(n_1^1, n_2^1$ and $n_1^2, n_2^2)$ do not complete.

## REFERENCES

ALTHEIM, M. AND MCCARRON, S. (EDS). 2000. XHTML 1.0: The extensible hyperText markup language. W3C Recommendation. Available at http://www.w3.org/TR/xhtml1/.

BERNERS-LEE, T. AND CONNOLLY, D. 1995. HyperText markup language specification 2.0. IETF RFC 1866.

BERSTEL, J. AND BOASSON, L. 2002. *Balanced Grammars and Their Languages*. Springer-Verlag New York, NY. 3–25.

BOAG, S., CHAMBERLIN, D., FERNÁNDEZ, M., FLORESCU, D., ROBIE, J., AND SIMÉON, J. (EDS). 2005. XQuery 1.0: An XML query language. W3C Candidate Recommendation. Available at http://www.w3.org/TR/2005/CR-xquery-20051103/.

BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., MALER, E., AND YERGEAU, F. (EDS). 2004. Extensible markup language (XML) 1.0 (3rd Ed.). W3C Recommendation. Available at http://www.w3.org/TR/2004/REC-xml-20040204/.

BRICKLEY, D. AND GUHA, R. V. (EDS). 2004. Vocabulary Description Language 1.0: RDF schema. W3C Recommendation. Available at http://www.w3.org/TR/rdf-schema.

BROWN, A., FUCHS, M., ROBIE, J., AND WADLER, P. 2001. MSL: A model for W3C XML schema. In *International World Wide Web Conference (WWW)*. Hong Kong.

BRÜGGEMANN-KLEIN, A. AND WOOD, D. 1998. One-unambiguous regular languages. *Inform. Computat. 140*, 2, 229–253.

BRÜGGEMANN-KLEIN, A. AND WOOD, D. 2004. Balanced context-free grammars, hedge grammars and pushdown caterpillar automata. In *Extreme Markup Languages*. Montreal, Canada.

BRZOZOWSKI, J. A. 1964. Derivatives of regular expressions. *J. ACM 11*, 4 (Oct.), 481–494.

BUNEMAN, P., DAVIDSON, S., FAN, W., HARA, C., AND TAN, W.-C. 2002. Keys for XML. *Comput. Netw. 39*, 473–487.

CHRISTENSEN, A. S., MOLLER, A., AND SCHWARTZBACH, M. I. 2003. Extending Java for high-level Web service construction. *ACM Trans. Program. Lang. Syst. 25*, 6 (Nov.), 814–875.

CLARK, J. 2001a. The design of RELAX NG. Available at http://www.thaiopensource.com/relaxng/design.html.

CLARK, J. 2001b. TREX—Tree regular expressions for XML. Available at http://www.thaiopensource.com/trex/.

CLARK, J. 2002. An algorithm for RELAX NG validation. Available at http://www.thaiopensource.com/relaxng/derivative.html.

CLARK, J. 2003. Namespace routing language. Available at http://www.thaiopensource.com/relaxng/nrl.html.

CLARK, J. AND MURATA, M. (EDS). 2001. RELAX NG specification. Available at http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

COMON, H., DAUCHET, M., GILLERON, R., JACQUEMARD, F., LUGIEZ, D., TISON, S., AND TOMMASI, M. 1997. Tree automata techniques and applications. Available at http://www.grappa.univ-lille3.fr/tata.

FALLSIDE, D. C. (EDS). 2001. XML Schema Part 0: Primer. W3C Recommendation. Available at http://www.w3.org/TR/xmlschema-0.

FORDIN, S. 2003. Java Architecture for XML Binding(JAXB) 1.0. JSR.

FRY, C. 2003. Streaming API for XML. JSR 173. Available at http://www.jcp.org/en/jsr/detail?id=173.

GLUSHKOV, V. 1961. The abstract theory of automata. *Russian Math. Surv. 16*, 1–53.

HOPCROFT, J. E. AND ULLMAN, J. D. 1979. *Introduction to Automata Theory, Language, and Computation*. Addison–Wesley.

HOSOYA, H. AND MURATA, M. 2002. Validation and boolean operations for attribute-element constraints. In *Programming Languages Technologies for XML (PLAN-X)*.

HOSOYA, H. AND MURATA, M. 2003. Boolean operations and inclusion test for attribute-element constraints. In *Proceedings of the 8th International Conference on Implementation and Application of Automata.* Santa Barbara, CA (July). Lecture Notes in Computer Science, vol. 2759. Springer, 201–212.

HOSOYA, H. AND PIERCE, B. C. 2002. Regular expression pattern matching for XML. *J. Function. Program. 13*, 6, 961–1004.

HOSOYA, H. AND PIERCE, B. C. 2003. XDuce: A typed XML processing language. *ACM Trans. Internet Tech. 3*, 2, 117–148.

HOSOYA, H., VOUILLON, J., AND PIERCE, B. C. 2003. Regular expression types for XML. *ACM Trans. Program. Lang. Syst. 27*, 1, 46–90.

ISO 8879. 1986. *Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML)*.

ISO/IEC. 2001. *Information Technology—Document Description and Processing Languages—Regular Language Description for XML (RELAX)—Part 1: RELAX Core*. ISO/IEC. TR 22250-1 2000(E).

ISO/IEC. 2003. *Information Technology—Document Schema Definition Languages (DSDL)—Part 2: Regular Grammar-Based Validation-RELAX NG*. ISO/IEC. 19757-2:2000E.

JEDRZEJOWICZ, J. AND SZEPIETOWSKI, A. 2001. Shuffle languages are in P. *Theoret. Comput. Science 250*, 1-2, 31–53.

JELLIFFE, R. 2000. Schematron. Web page. Available at http://www.ascc.net/xml/resource/schematron/.

JELLIFFE, R. (ED). 2005. Schematron. ISO/IEC FDIS 19757-3. Available at http://www.jtc1sc34.org/repository/ 0691.pdf.

KLARLUND, N., MOLLER, A., AND SCHWATZBACH, M. I. 1999. Document structure description 1.0. Specification. Available at http://www.brics.dk/DSD/dsddoc.html.

KLARLUND, N., MOLLER, A., AND SCHWATZBACH, M. I. 2000. DSD: A schema language for XML. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice*. Portland, OR.

KLARLUND, N., SCHWENTICK, T., AND SUCIU, D. 2003. XML: Model, schemas, types, logics and queries. In *Logics of Emerging Applications of Databases*. Springer Verlag.

KNUTH, D. E. 1973. *Fundamental Algorithms*. Addison-Wesley.

LEE, D. AND CHU, W. W. 2000. Comparative analysis of six XML schema languages. *ACM SIGMOD Record 29*, 3 (Sept.), 76–87.

MEGGINSON, D. 2000. SAX 2.0: The simple API for XML Web page. Available at http://www.megginson.com/SAX/index.html.

Milo, T., Suciu, D., and Vianu, V. 2000. Typechecking for XML transformers. In *ACM Principles Datab. Syst*. 11–22.

Murata, M. 1997. Transformation of documents and schemas by patterns and contextual conditions. In *Principles of Document Processing*. Vol. 1293. Springer-Verlag, 153–169.

Murata, M. 1999a. Regularity and locality of string languages and tree languages. Web page. Available at http://www.oasis-open.org/cover/murataRegularity.html.

Murata, M. 1999b. Syntax for regular-but-non-local schemata for structured documents. Web page. Available at http://www.oasis-open.org/cover/murataSyntax19990311.html.

Murata, M. 2000. RELAX (REgular LAnguage description for XML). Web page. Available at http://www.xml.gr.jp/relax/.

Murata, M., Lee, D., and Mani, M. 2001. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*. Montreal, Canada. Available at http://www.mulberrytech.com/Extreme/Proceedings/html/2001/Murata01/EML2001Murata01/html.

Murata, M. and Robie, J. 1999. Observations on structured document query languages. Available at http://www.w3.org/TandS/QL/QL98/pp/murata-san.html.

Murata, M. (Eds). 2002. RELAX namespace. ISO/IEC DTR 22250-2. Available at http://www.y-adagio.com/public/standards/iso_tr_relax_ns/dtr_22250-2.doc.

Murata, M. (Eds). 2005. *Namespace-based Validation Dispatching Language*. ISO/FDIS 19757-4. Available at http://www.jtc1sc34.org/repository/0694.pdf.

Neven, F. 2002. Automata Theory for XML Researchers. *ACM SIGMOD Record 31*, 3 (Sept.), 39–46.

OASIS. 1997–2003. SGML/XML and forest/hedge automata Theory. Web page. Available at http://xml.coverpages.org/hedgeAutomata.html.

Ogbuji, U. 2002. XML class warfare. Application Development Trends. Available at http://www.adtmag.com/article.asp?id=6965.

Raggett, D. 1997. HTML 3.2 Reference Specification. W3C Recommendation. Available at http://www.w3.org/TR/REC-html32/.

Raggett, D., Hors, A. L., and Jacobs, I. 1999. HTML 4.01 Specification. W3C Recommendation. Available at http://www.w3.org/TR/html4/.

Shin, K.-H. 1992. Some equivalnce between multi-level and single-level layout processes. In *the International Workshop on Principles of Document Processing (PODP)*.

Takahashi, M. 1975. Generalizations of regular sets and their applicatin to a study of context-free languages. *Inform. Control 27*, 1 (Jan.), 1–36.

Thompson, H. S., Beech, D., Maloney, M., and Mendelsohn, N. (Eds). 2001. XML Schema Part 1: Structures. W3C Recommendation. Available at http://www.w3.org/TR/xmlschema-1/.

Tozawa, A. 2001. Towards static type inference for XSLT. In *ACM Symposium on Document Engineering*.

Tozawa, A. and Hagiya, M. 2003. XML schema containment checking based on semi-implicit techniques. In *Proceedings of Implementation and Application of Automata, 8th International Conference, (CIAA 03)*, Santa Barbara, CA (July). O. H. Ibarra and Z. Dang, Eds. Lecture Notes in Computer Science, Vol. 2759. Springer, 213–225.

van der Vlist, E. 2002. Can XML be the same after W3C XML schema? XML.com. Available at http://www.xml.com/pub/a/2002/06/19/vdv-wxs.html.

Vianu, V. 2001. A Web odyssey: From Codd to XML. In *ACM Principles of Database Systems*. Santa Barbara, CA.

Walsh, N. and Muellner, L. 1999. *DocBook: The Definitive Guide*. O'Reilly & Associates.

Wood, L., Hors, A. L., Apparao, V., Byrne, S., Champion, M., Isaacs, S., Jacobs, I., Nicol, G., Robie, J., Sutor, R., and Wilson, C. (Eds). 1998. Document Object Model (DOM) Level 1 Specification Version 1.0. W3C Recommendation. Available at http://www.w3.org/TR/REC-DOM-Level-1/.

Received December 2003; revised August 2004, November 2004; accepted November 2004