Taxonomy of XML Schema Languages using Formal Language Theory

On the basis of regular tree languages [CDG+97] we present a mathematical framework for XML schema languages. This framework helps to formally describe, compare, and implement such XML schema languages. Our main results are as follows:

1. *Four subclasses of regular tree languages: local tree languages, single-type tree languages, restrained competition tree languages, and regular tree languages.*
2. *A classification and comparison of a few XML schema proposals and type systems:DTD, XML-Schema, DSD, XDuce, RELAX, and TREX*
3. *Properties of the grammar classes under two common operations: XML document validation and type assignment.*

Makoto Murata, IBM Tokyo Research Labs.
Dongwon Lee, UCLA / CSD
Murali Mani, UCLA / CSD

1. Introduction

As the popularity of XML increases substantially, the importance of XML schema language to describe the structure and semantics of XML documents also increases. Although there have been about a dozen XML schema language proposals made recently [LC00], no comprehensive mathematical analysis of such schema proposals has been available. We believe that providing a framework in abstract mathematical terms is important to understand various aspects of XML schema languages and to facilitate their efficient implementations.

Towards this goal, in this paper, we propose to use formal language theory, especially tree grammar theory, as such a framework. Given an XML document and its schema, suppose one wants to check whether the document is valid against the schema and further find out the types (or non-terminals) associated with each element in the document. We are interested in algorithms for such document validation and type assignment operations, and the time complexity of such algorithms. Furthermore, we would like to know if such algorithms can be implemented on top of SAX [Meg00] or rather require DOM [WHA+00]. Such issues are closely related with the efficient implementation of XML schema language proposals, and are directly addressed by our mathematical framework.

Our contributions are, in short, as follows:

1. we define four subclasses of regular tree grammars and their corresponding languages to describe XML content models precisely.
2. based on the framework, we present a detailed analysis and comparison of a few XML schema proposals and type systems; in our framework, it is straightforward to specify,

say, an XML schema proposal *A* is more powerful than an XML schema proposal *B* with respect to its expressive power of content models, and

3. we present results on the complexity of two important schema operations: document validation and type assignment (document interpretation).

We have seen so far that regular tree grammar theory provides an excellent framework for studying several aspects of XML schema languages. This is because the set of document structures defined by *any XML schema language* (a document structure is a tree with element tag names and attribute names, but without attribute values or text values) can be *precisely* described using this theory. However, there are some features in XML Schema languages that cannot be captured using this theory. For example:

- From a database perspective, a schema for a particular application specifies the *structure* (entities and relationships) and the *constraints* (such as key constraints) of the underlying data. A regular tree grammar specifies the entities and also parent-child relationships. However it cannot capture other kind of relationships represented using IDREF/(S), foreign keys etc. It also cannot represent integrity constraints (eg: the number of female employees should be at least 40% of the total number of employees).

1.1 Conventions

Historically, there are multiple equivalent terms used by different research sub-communities within XML. Three of the research sub-communities that are working on XML research and the terms used by them are listed below:

- The database and web community use terms such as schema and XML document. *Schema* is considered to be an abstract definition of the set of conforming *XML documents* [TBMM00]. Example is shown in below:

  book.xsd

```
<!DOCTYPE book [
  <!ELEMENT book (author+, publisher)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT publisher (EMPTY)>
  <!ATTLIST publisher Name CDATA #IMPLIED>
]>
```

  book.xml

```
<book>
  <author>J. E. Hopcroft</author>
  <author>J. D. Ullman</author>
  <publisher Name="Addison-Wesley"/>
</book>
```

- The formal language community uses terms such as grammar, instance tree, automata, and language. A *grammar* defines an abstract set of trees (i.e., *instance trees*), *language* denotes the set of trees accepted by a grammar, an *automaton* is used to check whether a

given instance tree belongs to a language [CDG+97]. We use *G* to denote the grammar, *L(G)* to denote the language, and *T (V, E)* (*V* is the set of nodes and *E* is the set of edges in *T*) to denote an instance tree. Example grammar is shown below:

```
N = {Book, Author, Publisher, Pcdata}
T = {book, author, publisher, pcdata}
S = {Book}
P = {Book → book(Author+, Publisher), Author → author (Pcdata),
     Publisher → publisher (ε), Pcdata → pcdata (ε)}
```

Instance tree corresponding to the grammar is shown in Figure 1.

- The type theory community uses terms such as type definitions, variables, and values. A set of *type definitions* is equivalent to a schema or grammar. An XML document is said to be a set of *values*, where each value can be viewed as being assigned to a *variable* [HVP00]. Example is shown below:

  type definitions

  ```
  type Book = book [Author+, Publisher]
  type Author = author [String]
  type Publisher = publisher [ ]
  ```

  variables and values

  ```
  Book b1 = book [a1, a2, p1]
  Author a1 = author ["J. E. Hopcroft"]
  Author a2 = author ["J. D. Ullman"]
  Publisher p1 = publisher [ ]
  ```

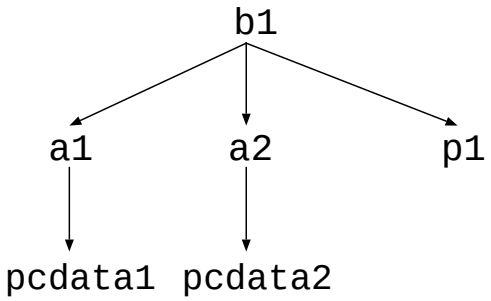

Figure 1.

Instance tree

We replace every text value in the XML document by a node called <pcdata> for convenience.

1.2 Related Work

According to [LC00], there have been about a dozen XML schema languages proposed lately. Previously, [Jel00],[LC00] attempt to compare and classify such XML schema proposals from various perspectives. However, their approaches are by and large not mathematical so that the precise description and comparison among XML schema proposals are not straightforward.

Our framework relies largely on regular tree grammars and languages as well as an important variation called regular hedge languages [Tak75], [Mur00a]. A description of tree languages and their closure properties are presented in [CDG+97]. Our contributions to this field are four subclasses of regular tree languages, and algorithms tailored for these subclasses.

1.3 Outline of the paper

The remainder of this paper is organized as follows. In Section 2, we first introduce regular tree languages and grammars, and then introduce restricted classes. In Section 3, we introduce validation algorithms for the four classes, and study the complexity of these algorithms. In Section 4, on the basis of these observations, we evaluate different XML schema language proposals. Finally, concluding remarks and thoughts on future research directions are discussed in Section 5.

2. Tree Grammars

Traditional context-free (string) grammars or regular (string) grammars [HU79] represent sets of *strings*. Successful parsing of strings against such grammars provides derivation trees. This scenario is appropriate for programming languages and natural languages, where programs and natural language text are strings rather than trees. On the other hand, start tags and end tags in an XML document collectively represent a *tree*. Construction of such trees requires XML documents only without relying on schemas written in XML schema languages such as DTD. Rather, schemas are used for validating constructed trees.

Since traditional context-free (string) grammars or regular (string) grammars are originally designed to describe permissible strings, they are inappropriate for describing permissible trees. In this section, as a mechanism for describing permissible trees, we define regular tree grammars. We then introduce three restricted classes called "local", "single-type", and "restrained-competition".

2.1 Regular Tree Grammars and Languages

We borrow the definitions of regular tree languages and tree automata in [CDG+97], but allow trees with ``infinite arity''; that is, we allow a node to have any number of subordinate nodes, and allow the right-hand side of a production rule to have a regular expression over non-terminals.

Definition 1. *(Regular Tree Grammar)* A regular tree grammar (RTG) is a 4-tuple $G = (N, T, S, P)$, where:

- N is a finite set of *non-terminal symbols*,

- *T* is a finite set of *terminal symbols*,
- *S* is a set of *start symbols*, where S ⊆ N.
- *P* is a finite set of production rules of the form X → a r, where X ∈ N, a ∈ T, and r is a regular expression over *N* (Remark: *X* is the *left-hand side*, a r is the *right-hand side*, and r is the *content model* of this production rule.)

Example 1. The following grammar G_1 = *(N, T, S, P)* is a regular tree grammar:

```
N = {Doc, Para1, Para2, Pcdata}
T = {doc, para, pcdata}
S = {Doc}
P = {Doc → doc (Para1, Para2*), Para1 → para (Pcdata),
     Para2 →para (Pcdata), Pcdata → pcdata ε}
```

Without loss of generality, we can assume that no two production rules have the same non-terminal symbol in the left-hand side and the same terminal symbol 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. We also assume that every non-terminal is either a start symbol or occurs in the content model of some production rule (this implies no useless non-terminals).

We have to define how a regular tree grammar generates a set of trees over terminal symbols. We first define interpretations.

Definition 2. *(Interpretation)* 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:

- $I(e_{root})$ is a start symbol where e_{root} is the root of *t*, and
- for each node *e* and its subordinates e_0 , e_1 , ..., e_i ,there exists a production rule X → a r such that
  - *I(e)* is X,
  - the terminal symbol (label) of *e* is *a*, and
  - $I(e_0)\ I(e_1)\ ...\ I(e_i)$ matches *r*.

Example 2. Consider a G_1 of Example 1 again. Suppose an XML document (or tree) <doc><para/><para/></doc> A mapping from the doc (i.e., root), the first para, and the second para to *Doc*, *Para1*, and *Para2*, respectively, is an interpretation of this tree against G_1.

Now, we are ready to define generation of trees from regular tree grammars.

Definition 3. *(Generation)* A tree *t* is generated from a regular tree grammar *G* if there is an interpretation of *t* against *G*. A set of trees is generated from *G* if each tree in this set is generated and no other trees are generated.

Example 3. The tree in Example 2 is generated from G_1 .

Definition 4. *(Regular Tree Language)* A tree language is a *regular tree language* if it is generated by a regular tree grammar.

2.2 Local Tree Grammars and Languages

We first define competition of non-terminals, which makes validation difficult. Then, we introduce a restricted class called ``local" by prohibiting competition of non-terminals [Tak75], [Mur99a], [Mur99b]. This class roughly corresponds to DTD.

Definition 5. *(Competing Non-Terminals)* Two different non-terminals *A* and *B* are said *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 symbol in the right-hand side.

Example 4. Consider a regular tree grammar $G_4 = (N, T, S, P)$, where:

```
N = {Book, Author1, Son, Article, Author2, Daughter}
T = {book, author, son, daughter}
S = {Book, Article}
P = {Book → book (Author1), Author1 → author (Son), Son → son ε,
     Article → article (Author2), Author2 → author (Daughter),
     Daughter →daughter ε}
```

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

Definition 6. *(Local Tree Grammar)* A *local tree grammar (LTG)* is a regular tree grammar that does not have competing non-terminals.

Example 5. G_4 in Example 4 is not local, since *Author1* and *Author2* compete with each other.

Example 6. The following grammar $G_6 = (N, T, S, P)$ is a local tree grammar:

```
N = {Book, Author1, Son, Pcdata}
T = {book, author, son, pcdata}
S = {Book}
P = {Book → book (Author1), Author1 → author (Son),
     Son → son Pcdata, Pcdata → pcdata ε}
```

Definition 7. *(Local Tree Language)* A tree language is a *local tree language* if it is generated by a local tree grammar.

Observe that local tree grammars and extended context-free (string) grammars (ECFG) look similar. However, the former describes sets of trees, while the latter describes sets of strings. The parse tree set of an ECFG is a local tree language.

2.3 Single-Type Tree Grammars and Languages

Next, we introduce a less restricted class called ``single-type'' by prohibiting competition of non-terminals within a single content model. This class roughly corresponds to XML-Schema.

Definition 8. *(Single-Type Tree Grammar)* A regular tree grammar is a *single-type tree grammar* if:

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

Example 7. The grammar G_1 shown in Example 1 is not single-type. Observe that non-terminals *Para1* and *Para2* compete, and they occur in the content model of the production rule for *Doc*.

Example 8. Consider G_4 in Example 4. This grammar is a single-type tree grammar since no production rules have more than one non-terminal in their content models and thus there cannot be any competing non-terminals in the content models.

Definition 9. *(Single-Type Tree Language)* A language is a single-type tree language if it is generated by a single-type tree grammar.

2.4 Restrained-Competition Tree Grammars and Languages

Now, we introduce an even less restricted class called ``restrained-competition''. The key idea is to use content models for restraining competition of non-terminals.

Definition 10. *(Restraining)* A content model *r* restrains competition of two competing non-terminals *A* and *B* if, for any sequences *U, V, W* of non-terminals, *r* do not generate both *U A V* and *U B W*.

Definition 11. *(Restrained-Competition Tree Grammar)* A regular tree grammar is a *restrained-competition tree grammar* if

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

Example 9. The grammar G_1 is a restrained-competition tree grammar. Non-terminals *Para1* and *Para2* 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 non-terminal and *Para2* may occur only as the non-first non-terminal.

Example 10. The following grammar G_{10} is not a restrained-competition tree grammar. Observe that the content model `(Para1*, Para2*)` does not restrain the competition of non-terminals *Para1* and *Para2*. For example, suppose that $U = V = W = \varepsilon$. Then, both *U Para1 V* and *U Para2 W* match the content model.

```
N = {Doc, Para1, Para2, Pcdata}
T = {doc, para, pcdata}
S = {Doc}
P = {Doc → doc (Para1*, Para2*), Para1 → para (Pcdata),
     Para2 → para (Pcdata), Pcdata → pcdata ε}
```

Definition 12. *(Restrained-Competition Tree Language)* A tree language is a *restrained-competition tree language* if it is generated by a restrained-competition tree grammar.

2.5 Summary of Examples

Each of the example grammars demonstrates one of the classes of tree grammars. The following table shows to which class each example belongs.

	Regular	**Restrained-Competition**	**Single-Type**	**Local**
G_6	Yes	Yes	Yes	Yes
G_4	Yes	Yes	Yes	No
G_1	Yes	Yes	No	No
G_{10}	Yes	No	No	No

2.6 Expressiveness and Closure

Having introduced four classes of grammars and languages, we here list some important observations. The interested reader is referred to [LMM00] for the proofs. We first present some observations on the expressive power of the four classes.

- Regular tree grammars are strictly more expressive than restrained-competition tree grammars. That is, some regular tree grammars cannot be rewritten as restrained-competition tree grammars.
- Restrained-Competition grammars are strictly more expressive than single-type tree grammars. That is, some restrained-competition tree grammars cannot be rewritten as single-type tree grammars.
- Single-type tree grammars are strictly more expressive than local tree grammars. That is, some single-type tree grammars cannot be rewritten as local tree grammars.

Next, we present observations on closure. A class of languages is said to be closed under union/intersection/difference when, for any two languages in that class, the union/intersection/difference of the two languages also belongs to the same class.

- The class of regular tree languages is closed under union, intersection, and difference.
- The class of local tree languages is closed under intersection, but is not closed under union or difference.
- The class of single-type tree languages is closed under intersection, but is not closed under union or difference.

- The class of restrained-competition languages is closed under intersection, but is not closed under union or difference.

3. Document Validation

In this section, we consider document validation. Given a grammar and a tree, we want to determine whether that tree is generated by the grammar, and also want to construct interpretations of the tree under the grammar.

3.1 Tree Model vs. Event Model

Programs for handling XML documents, including those for validation, are typically based on either the tree model or event model. In the tree model, the XML parser creates a tree in memory. Application programs have access to the tree in memory via some API such as DOM [WHA+00], and can traverse the tree any number of times. In the event model, the XML parser does not create a tree in memory, but merely raise events for start tags or end tags. Application programs are notified of such events via some API such as SAX [Meg00]. In other words, application programs visit and leave elements in the document in the depth-first manner. Once an application program visits an element in the document, it cannot visit the element again. Both models work fine for small documents. When documents are significantly large, the tree model has performance problems while the event model scales very well.

3.2 DTD validation and their variations

We first begin with validation for local tree grammars. Since DTDs correspond to local tree grammars, such validation has been used for years. Remember that a local tree grammar does not have competing non-terminals (see Definition 6). Thus, for each element, we can uniquely determine a non-terminal from the terminal symbol of this element.

Validation for local tree grammars can be easily built in the event model. Suppose that the XML processor has encountered a start tag for an element *e* in a given document. We can easily determine a non-terminal for *e* from this start tag. Next, suppose that the XML processors has encountered the end tag for *e*. Let $e_1, e_2, \ldots e_i$ be the child elements of *e*. Since the start tags for these elements have been already encountered by the XML parser, we can assume that we already know the non-terminals $n_1, n_2, \ldots n_i$ assigned to $e_1, e_2, \ldots e_i$. We only have to examine if $n_1, n_2, \ldots n_i$ matches the content model of the (unique) production rule for *e*.

This idea is effected by the following algorithm.

Algorithm 1: Validation for local tree grammars

1. **begin element:** When a start tag is encountered, we first determine a terminal symbol, say *a*. We then search for a production rule $X \rightarrow a\ r$ in the given grammar. It is guaranteed that one and at most one such production rule is found.
2. **end element:** When an end tag is encountered, the sequence of non-terminals assigned to the children of this element is compared against *r*. If this sequence does not match *r*, we report that this document is invalid and halt.

Observe that this algorithm not only determines whether a document is generated by the given grammar but also constructs an interpretation of the document. Since any document has at most one interpretation under a local tree grammar, the constructed interpretation is the only interpretation of the input document.

Now, we extend our algorithm for single-type tree grammars and restrained-competition tree grammars. This extension is quite simple and straightforward.

Remember that a single-type tree grammar does not allow competing non-terminals within a single content model (see Definition 8). Thus, for each element, we can uniquely determine a non-terminal from the terminal symbol of this element as well as the non-terminal for the parent element. Furthermore, since a single-type tree grammar does not allow competing start symbols, we can uniquely determine a non-terminal from the terminal symbol of the root element.

We only have to change the behavior for start tag events.

Algorithm 2: Validation for single-type tree grammars.

1. **begin element:** When a start tag is encountered, we first determine a terminal symbol, say *a*.
    1. **root:** If this start tag represents the root element, we search for a production rule $X \rightarrow a\ r$ in the given grammar such that *X* is a start symbol. It is guaranteed that at most one such production rule is found. If not found, we report that this document is invalid and halt.
    2. **non-root:** If this start tag represents a non-root element, we have already encountered the start tag for the parent element and have determined the non-terminal and content model, say *n'* and *r'*, for the parent element. We search for a production rule $X \rightarrow a\ r$ such that *X* occurs in *r'*. It is guaranteed that at most one such production rule is found. If not found, we report that this document is invalid and halt.
2. **end element:** The same as in Algorithm 1.

Next, we consider restrained-competition tree grammars. Remember that each content model in a restrained-competition tree grammar does not allow competing non-terminals to follow the same sequence of non-terminals (see Definition 11). Thus, for each element, we can uniquely determine a non-terminal from the terminal symbol of this element as well as the non-terminals for the parent element and the elder sibling elements.

We only have to change the behavior for start tag events for non-root elements.

Algorithm 3: Validation for restrained-competition tree grammars.

1. **begin element:** When a start tag is encountered, we first determine a terminal symbol, say *a*.

1. **root:** The same as in Algorithm 2.
2. **non-root:** If this start tag represents a non-root element, we have already encountered the start tag for the parent element and elder sibling elements. We thus have already determined the non-terminal and content model, say *n'* and *r'*, for the parent element, and have determined the non-terminals, say $n_1, n_2, \ldots n_i$ for the elder sibling elements. $X \rightarrow a\ r$ such that *r'* allows *X* to follow $n_1 n_2 \ldots n_i$. It is guaranteed that at most one such production rule is found. If not found, we report that this document is invalid and halt.

2. **end element:** The same as in Algorithm 1.

Observe that given a document, the extended algorithms construct an interpretation of the document. Any document has at most one interpretation under a single-type or restrained competition tree grammar.

3.3 Variations of Tree Automata

Before we study algorithms for any tree regular grammars, we consider variations of tree automata. Automata for regular tree grammars have been studied in the past and present [CDG+97], [Mur00a]. Four automata are known for checking tree membership: deterministic top-down, non-deterministic top-down, deterministic bottom-up, and non-deterministic bottom-up. Top-down automata assign states to superior elements and then subordinate elements; bottom-up automata assign states to subordinate elements and then superior elements. Deterministic automata assign a state to each element, while non-deterministic automata assign any number of states to each element.

We do not provide formal definitions of these tree automata, but would instead like to mention an important observation: non-deterministic top-down, deterministic bottom-up and non-deterministic bottom-up tree automata are equivalent and accept all regular tree languages, but deterministic top-down tree automata capture only a subclass of regular tree languages.

Algorithms 1, 2, and 3 are similar to deterministic top-down automata. However, deterministic top-down tree automata assign a state to an element without examining that element; they only examine the parent element and the state assigned to it. Because of this restriction, deterministic top-down tree automata are almost useless for XML. On the other hand, Algorithms 1, 2, and 3 examine an element before assigning a non-terminal (state) to it.

One approach for validation is to use deterministic bottom-up tree automata. 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 non-terminals of the grammar and then constructing a transition function for these states. Execution of thus constructed deterministic bottom-up tree automata is straightforward, which is the biggest advantage of this approach. However, in this paper, we do not consider this approach further for two reasons. First, creation of deterministic bottom-up tree automata is not easy and is beyond the scope of this paper. Second, straightforward implementation will lead to poor error message.

3.4 Non-deterministic algorithms for regular tree grammars

In this section, we show two algorithms for regular tree grammars. Both are based on non-deterministic tree automata.

Our first algorithm simulates non-deterministic top-down tree automata. It cannot be implemented in the event model but requires the tree model.

This algorithm is effected by a simple recursive procedure *validate*. Given a content model and a sequence of elements, *validate* compares this sequence against the content model and reports success or failure.

Algorithm 4.1: Validate

1. **input:** a content model `r` and a sequence $e_1\ e_2\ ...\ e_i$ of elements.

   output: success or failure

2. Switch statement

   1. Case 1: `r = ε` (the null string)

   If $e_1\ e_2\ ...\ e_i$ is an empty sequence (i.e., $i = 0$), this procedure succeeds. Otherwise, it fails.

   2. Case 2: `r = x` (a non-terminal)

   If $e_1\ e_2\ ...\ e_i \neq e_1$ (i.e., $i \neq 1$), this procedure fails. If $e_1\ e_2\ ...\ e_i = e_1$ (i.e., $i = 1$), we identify those production rules `X` $\rightarrow$ `a r'` such that `a` is the terminal symbol of e_1. For each of these production rules, we recursively invoke *validate* for `r'` and $e_{11}\ e_{12}\ ...\ e_{1j}$, where $e_{11}\ e_{12}\ ...\ e_{1j}$ is the children of e_1. If at least one of these invocations succeeds, this procedure succeeds. Otherwise, it fails.

   3. Case 3: `r = r1 | r2`

   We invoke *validate* for `r1` and $e_1\ e_2\ ....\ e_i$ and then invoke it for `r2` and $e_1\ e_2\ ....\ e_i$. If at least one of the two invocations succeeds, this procedure succeeds. Otherwise, it fails.

   4. Case 4: `r = r1 r2`

   For each $k\ (1 \leq k \leq i)$, we invoke *validate* for `r1` and $e_1\ e_2\ ...\ e_k$ and also invoke *validate* for `r2` and $e_{k+1}\ e_{k+2}\ ...\ e_i$. If both invocations succeed for some k, this procedure succeeds. Otherwise, it fails.

   5. Case 5: `r = r1*`

   If $e_1\ e_2\ ...\ e_i = \varepsilon$ (i.e., $i = 0$), this procedure succeeds. Otherwise, for each $k\ (1 \leq k \leq i)$, we invoke *validate* for `r1` and $e_1\ e_2\ ...\ e_k$ and also invoke *validate* for `r1*` and $e_{k+1}\ e_{k+2}\ ...\ e_i$. If both invocations succeed for some k, this procedure succeeds. Otherwise, it fails.

Algorithm 4.2: Non-deterministic top-down validation for regular tree grammars.

1. We begin with the root element. For each start symbol `x`, we invoke Algorithm 4.1 **validate** for `x` and the root element. If at least one of these invocations succeeds, this document is valid. Otherwise, it is invalid.

This approach does not require understanding of formal language theory. On the other hand, this approach has significant disadvantages. First, this algorithm may cause exhaustive search. Second, this algorithm leads to poor error message. Given an invalid document, this algorithm tries all possibilities in turn. If all of them fail, this algorithm cannot tell which of the possibilities is closest to success and thus cannot report what is a required change.

Our second algorithm simulates non-deterministic bottom-up tree automata. It is also a significant extension of Algorithm 1; the biggest difference is that a set of non-terminals (rather than a single non-terminal) is assigned to each element. Although this extension is more advanced than Algorithms 2 and 3, it can still be built on top of the event model.

Algorithm 5: Validation for regular tree grammars

1. **begin element:** When a start tag is encountered, we identify those production rules `x` $\rightarrow$ `a r` such that *a* is the terminal symbol of this tag. Note that more than one production rule may be found.
2. **end element:** When an end tag is encountered, we have already encountered the end tags for the children $e_1, e_2, \ldots e_i$ of this element. We create non-terminal sequences by choosing one of the non-terminal assigned to each $e_j (j \leq i)$, and concatenating the chosen non-terminals. If at least one of the created non-terminal sequences match *r* (the content model of `x` $\rightarrow$ `a r`), then *X* is one of the non-terminals for this element. If none of the created non-terminals match any content model for this element, we report that this document is invalid and halt.

   When this element is the root element, validation succeeds if and only if at least one of the non-terminals assigned to the root element is a start symbol.

Observe that this algorithm does *not* provide a unique interpretation of the document. In fact, more than one interpretation may exist for a given tree. For example, suppose that an XML document `<doc><para/></doc>` is validated against the grammar in Example 10. Two non-terminals, namely `Para1` and `Para2`, are assigned to the element `<para/>`.

4. Evaluating Different XML Schemata Proposals

In this section, we compare six representative XML schema language proposals: DTD, DSD, XML-Schema, XDuce, RELAX and TREX. Our focus is on the mathematical properties of these schema languages in our framework, though we also mention other features such as ease of use whenever possible.

All these five schema proposals define a regular tree grammar. To help understand and analyze grammars better, in this section, we represent a regular tree grammar as a 6-tuple *G = (N1, N2, T, S, P1, P2)*. Here *N1* represents non-terminal symbols that produce trees, their production

rules are described in P1, these rules are of the form $A \rightarrow a\ X$, where $A \subseteq N1$, $a \subseteq T$, $X \subseteq N2$. *N2* represents non-terminal symbols that produce a list of trees, their production rules are described in P2, these rules are of the form $X \rightarrow RE$, where *RE* is a regular expression over $N1 \cup N2$, *T* is the set of terminal symbols, and $S \subseteq N1$ is the set of start symbols.

Example 11. The grammar G_1 of Example 1 can be re-written in the above notation as G_{11} = *(N1, N2, T, S, P1, P2)*:

```
N1 = {Doc, Para1, Para2, Pcdata}
N2 = {DOC, PARA1, PARA2, PCDATA}
T  = {doc, para, pcdata}
S  = {Doc}
P1 = {Doc → doc DOC, Para1 → para PARA1,
      Para2 →para PARA2, Pcdata → pcdata PCDATA}
P2 = {DOC → (Para1, Para2*), PARA1 → (Pcdata),
      PARA2 → (Pcdata), PCDATA → ε}
```

The relationship between the expressive power of the various grammar classes in the previous section helps to compare the different XML Schema proposals (see Figure 2).

4.1 DTD

DTD as defined in [BPS98] is a local tree grammar. This is enforced by not distinguishing between terminal symbols and non-terminal symbols.

4.2 DSD

The main features of DSD [KMS00] are the use of *non-terminals*, *content expressions* to specify unordered content, *context patterns* which can be used to specify structures based on other structures and also values, and specifying relationships using *points-to* for IDREF attributes. Using our framework, we can capture all the features except structure specification based on values and *points-to* for IDREF attributes.

Let us consider the structural specification by DSD: DSD does not impose any constraint on the production rules, therefore we can express any regular tree grammar in DSD. For example, E = *(*$Author_1$**, *$Publisher_1$**,*$Author_2$**)* is a perfectly valid content model in DSD, where $Author_1$ and $Author_2$ are competing non-terminals. But the parsing in DSD uses a greedy technique (no backtracking) with one vertical and horizontal lookaheads. Therefore DSD does not accept all regular tree languages. For example, DSD cannot accept a string of the form *(*$Author_1$*,* $Author_2$*)* which conforms to *E*.

Let us consider element definitions in DSD. An example element definition in DSD has the form:

```
<ElementDef ID="book_title" Name="title">
    SomeContentSpecification
</ElementDef>
```

This can be converted into the grammar notation as: `P1 = {book_title → title Expression}`. Here, *Expression* is equivalent to `SomeContentSpecification`.

4.3 XML-Schema

XML-Schema [TBMM00] represents a single type tree grammar. The main features of XML Schema are *complex type definitions*, *anonymous type definitions*, *group definitions*, *subtyping by extension and restriction*, *substitution groups*, *abstract type definitions* and *integrity constraints such as key, unique and keyref constraints*. The different concepts in XML-Schema can be described in our framework as illustrated below.

1. A complex type definition defines a production rule in *P2*. For instance,

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

   This can be converted into a P2 rule Book → (title, author+, publisher?), where Book ∈ N2 and {title, author, publisher} ⊆ N1.

2. A group definition defines a new non-terminal in *N2* as follows ([Fal00] Section 2.7):

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

   This group definition is equivalent to the following grammar rules.

```
P1 = {ShipTo → shipTo (USAddress), BillTo → billTo (USAddress)}
P2 = {shipAndBill → (ShipTo, BillTo)}
```

3. From object oriented programming, XML-Schema borrows the concepts of sub-typing. This is achieved through extension or restriction. An example of derived types by extension (slightly modified from [Fal00] Section 4.2) is given below.

```
<complexType name="Address">
  <sequence>
    <element name="name"/>
    <element name="street"/>
    <element name="city"/>
  </sequence>
</complexType>

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

```
        <sequence>
          <element name="postcode"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>
```

The above type definitions for Address and UKAddress is equivalent to the rules:

```
P2 = {Address → (name, street, city),
      UKAddress → (name, street, city, postcode)}
```

Now suppose there was an element declaration such as: `<element name="shipTo" type="Address"/>` This is equivalent to the following two rules: `P1 = {ShipTo → shipTo (Address), ShipTo → shipTo (UKAddress)}` Note that the rule `ShipTo → shipTo UKAddress` is automatically inserted, true to the object-oriented programming paradigm. But this can be considered as a ``side-effect'' in formal language theory. XML-Schema provides an attribute called block to prevent such side-effects. For example, if Address had defined *block=#all*, then we would not automatically insert the rule.

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

4. XML-Schema provides a mechanism called `xsi:type` which allows it to not satisfy the constraints of a restrained-competition tree grammar. For example, it is legal to have the following rules in XML-Schema.

```
P2 = {X → (Title), Y → (Title, Author1+), Z → (Title, Author2+, Publisher)}
P1 = {Book → book X, Book → book Y, Book → book Z}
```

The above grammar is not a restrained competition tree grammar. But for checking document validity, the properties of a single-type tree grammar are maintained by requiring that the document mention explicitly the type. For example a valid node in the instance tree would be: `<book xsi:type="Y">`.

It should be noted that this does not make XML-Schema equivalent to a regular tree grammar, because XML Schema cannot define a type such as `BOOK → (Title, Author1*, Author2*)` with `Author1→ author A1`, `Author2 → author A2`.

5. A substitution group definition (previously known as equivalence classes) can be converted into an equivalent grammar definition as follows. For example, consider the substitution group definition ([Fal00] Section 4.6) (We modify it slightly for easy explanation):

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

This is converted into grammar rules as:

```
P1 = {ShipComment → shipComment Y, customerComment → cutomerComment Z
```

```
Ipo:comment → shipComment Y, Ipo:comment → shipComment Z}
```

where *Ipo:comment*, *ShipComment* and *CustomerComment* are non-terminals in *N1*. Using such substitution groups require that there be a rule in *P1* of the form: `Ipo:comment → ipo:comment X`, and that *Y*, *Z* be derived from *X*.

4.4 XDuce

XDuce [HVP00] provides type definitions equivalent to regular tree grammars. The key features of XDuce are their *type constructors* where we can define regular expression types as derived from other types, and subtyping using *subtagging*.

A type definition in XDuce that produces a tree is converted in our framework into a rule in *P1*. Consider the example from [HVP00]: `type Addrbook = addrbook [Person*]` is written as `Addrbook → addrbook (Person*)`. Any type definition that does not produce a tree is written as *P2* rules. For example, `type X = T, X | ()` represents the *P2* rules `X → (T, X + ε)`. Note that XDuce writes the above type rules in a right-linear form, which makes every content model definition equivalent to a regular string language.

XDuce provides a mechanism called *subtagging*, which is a reflexive and transitive relation on terminal symbols in *T*. For example, consider the subtag declaration

```
subtag i <: fontstyle
```

To convert this subtagging relation into grammar, consider all *P1* rules that have `fontstyle` on the right hand side. Let these rules be `{A → fontstyle A1, B → fontstyle B1, ..., N → fontstyle N1}`. Now the subtag declaration adds the following additional rules to `P1: {A → i A1, B → i B1, ..., N → i N1}`

4.5 RELAX

Any regular tree grammar can be expressed in RELAX [Mur00b]. The main features of RELAX are `elementRules` and `hedgeRules`, and clear differentiation between them; these correspond directly to *P1* and *P2* respectively in our framework.

To convert RELAX to our framework, let us first consider `elementRule`, `hedgeRule` and `tag` elements, but where `tag` elements do not specify a `role` attribute. In this case, the value of the `name` attribute of the `tag` element is considered to be the value of the `role` attribute.

1. An `elementRule` with a corresponding `tag` element defines a *P1* rule. For example consider the `elementRule`, slightly modified from the one in [Mur00b].

```
<elementRule role="section" label="Section">
  <ref label="paraWithFNotes" occurs="*"/>
</elementRule>

<tag name="section"/>
```

This can be converted into the production rules: `P1 = {Section → section (paraWithFNotes*)}`

2. `hedgeRule` defines a *P2* rule. For example, consider the `hedgeRule` mentioned in [Mur00b]

```
<hedgeRule label="blockElem">
  <ref label="para"/>
</hedgeRule>
```

The above `hedgeRule` can be converted into a grammar as: `P2 = {blockElem → (para)}`

RELAX allows a `hedgeRule` to reference to another `hedgeRule`. But it requires that there be no recursion in the `hedgeRules`; this ensures that the grammar remains regular.

3. RELAX allows multiple `hedgeRules` to share the same label. For example, we can specify in RELAX two `hedgeRules` as

```
<hedgeRule label="blockElem">
  <ref label="para"/>
</hedgeRule>
<hedgeRule label="blockElem">
  <ref label="itemizedList"/>
</hedgeRule>
```

This can be converted into *P2* rules as:`P2 = {blockElem → (para), blockElem → (itemizedList)}`

4. RELAX allows multiple `elementRules` to share the same label as follows. For example, we can have

```
<elementRule role="a" label="A">
  <ref label="X"/>
</elementRule>
<elementRule role="a" label="A">
  <ref label="Y"/>
</elementRule>
<elementRule role="b" label="A">
  <ref label="Z"/>
</elementRule>

<tag name="a"/>
```

These three `elementRules` can be converted into three rules as: `P1 = {A → a X, A → a Y, A → b Z}`

5. Now let us consider when `tag` elements specify a `role` attribute. In effect this adds an additional level of indirection in rule specification. To convert into the grammar representation, we need to collapse the role attributes. This will be clear from the following example [Mur00b].

```
<tag name="val" role="val-integer"/>
<elementRule role="val-integer" label="Val">
    <ref label="X"/>
</elementRule>
<tag name="val" role="val-string"/>
<elementRule role="val-string" label="Val">
  <ref label="Y"/>
</elementRule>
```

They can be converted into grammar rules as: `P1 = {Val → val X, Val → val Y}`

4.6 TREX

TREX [Cla01] is quite similar to RELAX. The main differences of TREX from RELAX are additional features for user convenience - for example, elements and attributes can be combined without restriction in content models, and an improved algorithm for validation.

TREX supports the notion of non-terminals using `define`. However, unlike RELAX, TREX does not distinguish between non-terminals that produce trees and non-terminals that produce list of trees (i.e., `N1` and `N2`). For example, consider the following TREX pattern:

```
<grammar>

  <start>
    <ref name="AddressBook"/>
  </start>

  <define name="AddressBook">
    <element name="addressBook">
      <zeroOrMore>
        <ref name="Card"/>
      </zeroOrMore>
    </element>
  </define>

  <define name="inline">
    <zeroOrMore>
      <choice>
        <element name="bold">
          <ref name="inline"/>
        </element>
        <element name="italic">
          <ref name="inline"/>
        </element>
      </choice>
    </zeroOrMore>
  </define>

</grammar>
```

Here `AddressBook` is a non-terminal that produces a tree, and `inline` is a non-terminal that produces a list of trees. The above TREX pattern will be represented in our framework as follows:

```
P1 = {AddressBook → addressBook ADDRESSBOOK, Bold → bold inline,
      Italic → italic inline}
P2 = {ADDRESSBOOK → (Card*), inline → (Bold | Italic)*}
```

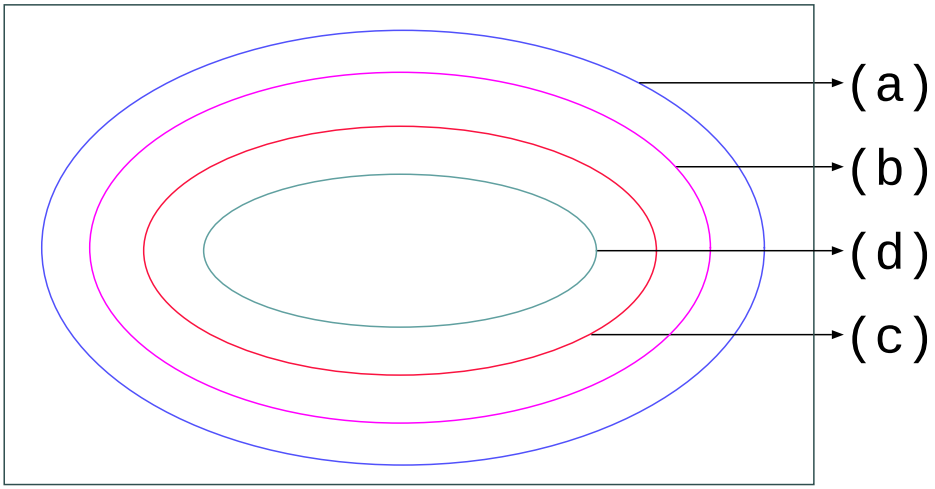

Figure 2.

The expressive power of the different regular tree grammars: (a) regular tree grammars (RELAX, XDuce, TREX), (b) restrained-competition tree grammars (c) single-type tree grammars (XML-Schema) (d) local tree grammars (DTD)

4.7 Implementations

In this subsection, we describe status of implementations of XML schema languages.

1. **DTD:** Membership checking against DTDs has been widely implemented. Most implementations are based on the event model, and use Algorithm 1.
2. **XDuce:** Type assignment implemented by the designers of XDuce is based on the tree model. This implementation is similar to Algorithm 4.
3. **DSD:** Type assignment of DSD is similar to that of XDuce. It is based on the tree model, and is a variation of Algorithm 4. However, backtracking is not thorough, and thus does not perform exhaustive search.
4. **XML-Schema:** Schema-validity assessment, as defined in 7.2 of XML Schema Part 1, is similar to Algorithm 2. To the best of our knowledge, all implementations follow this reference model.

1. **XSV:** XSV (XML Schema Validator) is based on the tree model.
2. **XML-Schema Processor:** This implementation can be combined with a SAX parser as well as a DOM parser. When it is combined with an event model parser it does not support `key`, `unique`, and `keyref` as of now (2001/3/31).
3. **Xerces Java Parser:** This implementation is based on the event model, and does not support `key`, `unique`, and `keyref` as of now (2001/3/31).
4. **XMLSpy & XMLInstance:** They are XML document editors, and thus use the tree model. Editing of documents can be controlled by schemas in XML Schema.

5. **RELAX:** Four implementations of RELAX are available at *http://www.xml.gr.jp/relax/*. They use different algorithms for type assignment and membership checking.
    1. **VBRELAX:** This program is based on the tree model. It simulates top-down non-deterministic automata by backtracking. Such backtracking may require exhaustive search.
    2. **RELAX Verifier for C++:** This program is based on the event model. It performs membership checking, but does not perform type assignment. This program is based on Algorithm 5..
    3. **RELAX Verifier for Java:** This program is based on the event model. It is also based on Algorithm 5, but further introduces top-down non-determinism. Intuitively speaking, when an element is visited, possible non-terminals for this element are temporarily assigned to the element. Such possible non-terminals are chosen by examining the parent element and the non-terminals temporarily assigned to it. When an element is left, this program examines which of the temporarily assigned non-terminals are allowed by the subordinate elements. Advantages of combining top-down non-determinism and bottom-up non-determinism are appropriate error messages and error recovery. RELAX Verifier performs type assignment as well as membership checking. Two linear-time algorithms for type assignment have been implemented. One requires two-pass processing of XML documents, and the other is dedicated to restrained-competition grammars without the deterministic constraint.
    4. **RELAXER:** Relaxer is a Java class generator. Given a RELAX module, Relaxer generates Java classes that represent XML documents permitted by the module. These Java classes receive XML documents as DOM trees and perform type assignment. This type assignment uses top-down non-deterministic automata by (limited) backtracking.

6. **TREX:** One implementation of TREX is available. The key feature of this algorithm is the efficiency and error message.
    1. **TREX Implementation by James Clark:**This implementation is a variation of Algorithm 5. However the efficiency of the algorithm lies in the handling of non-

determinism. This algorithm is based on derivatives of regular expressions, and lazy construction of automaton.

5. Conclusion

A mathematical framework using formal language theory to compare various XML schema languages is presented. This framework enables us to define various subclasses of regular tree languages, and study the closure properties and expressive power of these languages. Also, algorithms for document validation and type assignment for the different grammar classes are described. Finally, various schema language proposals are compared using our framework, and the implementations available are discussed.

Our framework brings forward a very important question: Do we need to migrate from deterministic content models of XML 1.0 in favor of schema languages such as RELAX and TREX that allow non-deterministic content models? Our work in this paper as well as other work, with regard to document processing and XML query, makes us believe that we should allow non-deterministic content models in XML schema languages.

We have multiple directions for future research which we are pursuing presently. We are examining ambiguity in regular tree grammars and languages, and studying how to determine whether a given regular tree grammar is ambiguous or not. We are also examining integrity constraints necessary for a schema language as studied widely in the area of database systems, and examining efficient implementations of these constraints for XML applications.

Bibliography

[BPS98] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds). ``Extensible Markup Language (XML) 1.0", Feb. 1998. *http://www.w3.org/TR/1998/REC-xml-19980210*.

[Cla01] J. Clark ``TREX - Tree Regular Expressions for XML", 2001. *http://www.thaiopensource.com/trex*.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. ``Tree Automata Techniques and Applications", 1997. *http://www.grappa.univ-lille3.fr/tata*.

[Fal00] D. C. Fallside (Eds). ``XML Schema Part 0: Primer", Oct. 2000. *http://www.w3.org/TR/xmlschema-0*.

[HU79] J. E. Hopcroft and J. D. Ullman. ``Introduction to Automata Theory, Language, and Computation". Addison-Wesley, 1979.

[HVP00] H. Hosoya, J. Vouillon, and B. C. Pierce. ``Regular Expression Types for XML". In Int'l Conf. on Functional Programming (ICFP), Montreal, Canada, Sep. 2000.

[Jel00] R. Jelliffe. ``The XML Schema Specification in Context", Feb. 2000. *http://www.ascc.net/~ricko/XMLSchemaInContext.html*.

[KMS00] N. Klarlund, A. Moller, and M. I. Schwatzbach. ``DSD: A Schema Language for XML". In ACM SIGSOFT Workshop on Formal Methods in Software Practice, Portland, OR, Aug. 2000.

[LC00] D. Lee and W. W. Chu. ``Comparative Analysis of Six XML Schema Languages". ACM SIGMOD Record, 29(3):76--87, Sep. 2000.

[LMM00] D. Lee, M. Mani, and M. Murata. ``Reasoning about XML Schema Languages using Formal Language Theory". Technical Report, IBM Almaden Research Center, RJ# 10197, Log# 95071, Nov. 2000. *http://www.cs.ucla.edu/~dongwon/paper*.

[Meg00] D. Megginson. ``SAX 2.0: The Simple API for XML", May. 2000. *http://www.megginson.com/SAX/index.html*.

[Mur99a] M. Murata. ``Regularity and Locality of String Languages and Tree Languages", Feb. 1999. *http://www.oasis-open.org/cover/murataRegularity.html*.

[Mur99b] M. Murata. ``Syntax for regular-but-non-local schemata for structured documents", Apr. 1999. *http://www.oasis-open.org/cover/murataSyntax19990311.html*.

[Mur00a] M. Murata. ``Hedge Automata: a Formal Model for XML Schemata", 2000. *http://www.xml.gr.jp/relax/hedge_nice.html*.

[Mur00b] M. Murata. ``RELAX (REgular LAnguage description for XML)", Aug. 2000. *http://www.xml.gr.jp/relax*.

[Tak75] M. Takahashi. ``Generalizations of Regular Sets and Their Applicatin to a Study of Context-Free Languages". Information and Control, 27(1):1--36, Jan. 1975.

[TBMM00] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn (Eds). ``XML Schema Part 1: Structures", Oct. 2000. *http://www.w3.org/TR/xmlschema-1/*.

[WHA+00] L. Wood, A. Le Hors, V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, G. Nicol, J. Robie, R. Sutor, and C. Wilson (Eds). ``Document Object Model (DOM) Level 1 Specification", Sep. 2000. *http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/*.

Biography

Makoto Murata
IBM Tokyo Research Labs.
E-mail: mmurata@trl.ibm.co.jp

Researcher working on document processing.

Dongwon Lee
UCLA / CSD

E-mail: dongwon@cs.ucla.edu

Ph.D candidate majoring Database with a focus on XML.

Murali Mani
UCLA / CSD
E-mail: mani@cs.ucla.edu

Ph.D candidate majoring Database with a focus on XML.