Taxonomy of XML Schema Languages using Formal Language Theory

On the basis of regular tree languages, we present a formal framework for XML schema languages. This framework helps to describe, compare, and implement such schema languages. Our main results are as follows: (1) four classes of tree languages, namely "local", "single-type", "restrained competition" and "regular"; (2) document validation algorithms for these classes; and (3) classification and comparison of schema languages: DTD, XML-Schema, DSD, XDuce, RELAX Core, and TREX.

Makoto Murata, IBM Tokyo Research Lab./ International University of Japan
Dongwon Lee, UCLA / CSD
Murali Mani, UCLA / CSD

1. Introduction

XML is a meta language for creating markup languages. To represent a particular type of information, we create an XML-based language by designing an inventory of names for elements and attributes. These names are then utilized by application programs dedicated to this type of information.

A *schema* is a description of such an inventory: a schema specifies permissible names for elements and attributes, and further specifies permissible structures and values of elements and attributes. Advantages of creating a schema are as follows: (1) the schema precisely describes permissible XML documents, (2) computer programs can determine whether or not a given XML document is permitted by the schema, and (3) we can use the schema for creating application programs (by generating code skeletons, for example). Thus, schemas play a very important role in development of XML-based applications.

Several languages for writing schemas, which we call *schema languages*, have been proposed in the past. Some languages (e.g., DTD) are concerned with XML documents in general; that is, they handle elements and attributes. Other languages are concerned with particular type of information which *may* be represented by XML; primary constructs for such information are not elements or attributes, but rather constructs specific to that type of information. RDF Schema of W3C is an example of such a schema language. Since primary constructs for RDF information are resources, properties, and statements, RDF Schema is concerned with resources, properties, and statements rather than elements and attributes. In this paper, we limit our concern to schema languages for XML documents in general (i.e., elements and attributes); specifically, we consider DTD [BPS00], XML-Schema [TBMM00], DSD [KMS00], RELAX Core [Mur00b], and TREX [Cla01]. [1] Although schema languages dedicated to particular types of information (e.g., RDF, XTM, and ER) are useful for particular applications, they are outside the scope of this paper.

We believe that providing a formal framework is crucial in understanding various aspects of schema languages and facilitating efficient implementations of schema languages. We use regular tree grammar theory ([Tak75] and [CDG+97]) to formally capture schemas, schema languages, and document validation. Regular tree grammars have recently been used by many

researchers for representing schemas or queries for XML and have become the mainstream in this area (see OASIS [OA01] and Vianu [VV01]); in particular, XML Query [CCF+01] of W3C is based on tree grammars.

Our contributions are as follows:

1. We define four subclasses of regular tree grammars and their corresponding languages to describe schemas precisely;
2. We show algorithms for validation of documents against schemas for these subclasses and consider the characteristics of these algorithms (e.g., the tree model .vs. the event model); and
3. Based on regular tree grammars and these validation algorithms, we present a detailed analysis and comparison of a few XML schema proposals and type systems; an XML schema proposal *A* is more expressive than another proposal *B* if the subclass captured by *A* properly includes that captured by *B*.

The remainder of this paper is organized as follows. In Section 2, we consider related works such as other survey papers on XML schema languages. In Section 3, we first introduce regular tree languages and grammars, and then introduce restricted classes. In Section 4, we introduce validation algorithms for the four classes, and consider their characteristics. In Section 5, 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 6.

2. Related Work

More than ten schema languages for XML have appeared recently, and [Jel00] and [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 schema language proposals are not straightforward. On the other hand, this paper first establishes a formal framework based on regular tree grammars, and then compares schema language proposals.

Since Kilho Shin advocated use of tree automata for structured documents in 1992, many researchers have used regular tree grammars or tree automata for XML (see OASIS [OA01] and Vianu [VV01]). However, to the best of our knowledge, no papers have used regular tree grammars to classify and compare schema language proposals. Furthermore, we introduce subclasses of regular tree grammars, and present a collection of validation algorithms dedicated to these subclasses.

XML-Schema Formal Description (formerly called MSL [BFRW01]) is a mathematical model of XML-Schema. However, it is tailored for 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 fine details of each schema language are not.

3. Tree Grammars

In this section, as a mechanism for describing permissible trees, we study tree grammars. We begin with a class of tree grammars called "regular", and then introduce three restricted classes called "local", "single-type", and "restrained-competition".

Some readers might wonder why we do not use context-free (string) grammars. Context-free (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*. Since traditional context-free (string) grammars are originally designed to describe permissible strings, they are inappropriate for describing permissible trees.

3.1 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-terminals*,
- T is a finite set of *terminals*,
- S is a set of *start symbols*, where S is a subset of N.
- P is a finite set of production rules of the form $X \rightarrow a\ r$, where $X \in N$, $a \in T$, and r is a regular expression over N; 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. The left-hand side, right-hand side, and content model of the first production rule is `Doc`, `Doc (Para1, Para2*)`, and `(Para1, Para2*)`, respectively.

```
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}
```

We represent every text value by the node `pcdata` for convenience.

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

We have to define how a regular tree grammar generates a set of trees over terminals. 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 (label) of e is a, and
  - $I(e_0)\ I(e_1)\ ...\ I(e_i)$ matches r.

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

Definition 3. *(Generation)* A tree t is generated by a regular tree grammar G if there is an interpretation of t against G.

Example 2. An instance tree generated by G_1 , and its interpretation against G_1 are shown in Figure 1.

Figure 1.

An instance tree generated by G_1, and its interpretation against G_1. We use unique labels to represent the nodes in the instance tree.

Definition 4. *(Regular Tree Language)* A *regular tree language* is the set of trees generated by a regular tree grammar.

3.2 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]. 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 in the right-hand side.

Example 3. Consider a regular tree grammar $G_3 = (N, T, S, P)$, 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 *author* in the right-hand side. There are no other competing non-terminal pairs in this grammar.

Definition 6. *(Local Tree Grammar and Language)* A *local tree grammar (LTG)* is a regular tree grammar without competing non-terminals. A tree language is a *local tree language* if it is generated by a local tree grammar.

Example 4. G_3 in Example 3 is not local, since *Author1* and *Author2* compete with each other.

Example 5. The following grammar $G_5 = (N, T, S, P)$ 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}
```

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.

3.3 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 7. *(Single-Type Tree Grammar and Language)* A a *single-type tree grammar* is a regular tree grammar such that

- 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.

A tree language is a single-type tree language if it is generated by a single-type tree grammar.

Example 6. 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 7. Consider G_3 in Example 3. 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.

3.4 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 8. *(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 9. *(Restrained-Competition Tree Grammar and Language)* A *restrained-competition tree grammar* is a regular tree grammar such that

- 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.

A tree language is a *restrained-competition tree language* if it is generated by a restrained-competition tree grammar.

Example 8. 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 9. The following grammar G_9 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}
```

3.5 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_5	Yes	Yes	Yes	Yes
G_3	Yes	Yes	Yes	No
G_1	Yes	Yes	No	No
G_9	Yes	No	No	No

3.6 Expressiveness and Closure

Having introduced four classes of grammars and languages, we consider expressiveness and closure. The interested reader is referred to [LMM00] for the proofs.

First, we consider expressiveness using practical examples. For each terminal, a local tree grammar provides a single content model for all elements of this terminal. For example, `<title>` elements may be subordinate to `<section>`, `<chapter>`, or `<author>` elements, but permissible subordinate children are always the same. A single-type tree grammar does not have this tight restriction, but cannot allow the first paragraph and the other paragraphs in a section to have different content models. A restrained-competition tree grammar lifts this restriction. Although some regular tree languages cannot be captured by restrained-competition tree grammars, we are not aware of any practical examples.

Next, we present theoretical observations about expressiveness.

- 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 tree 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.

4. Document Validation

In this section, we consider algorithms for document validation. Given a grammar and a tree, these algorithms determine whether that tree is generated by the grammar, and also construct interpretations of the tree. All algorithms are based on variations of tree automata. Some algorithms require the tree model while others do not.

4.1 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. While both models work fine for small documents, the tree model has performance problems for significantly large documents.

4.2 DTD validation and their variations

We 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 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, say *a*. We then search for a production rule `X → 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 7). Thus, for each element, we can uniquely determine a non-terminal from the terminal 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 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, say *a*.
    1. **root:** If this start tag represents the root element, we search for a production rule `X → 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 → 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 9). Thus, for each element, we can uniquely determine a non-terminal from the terminal 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, 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, ..., n_i for the elder sibling elements. We search for a production rule $X \rightarrow a\ r$ 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.

4.3 Variations of Tree Automata

Before we study algorithms applicable to arbitrary regular tree grammars, we consider tree automata. Validation of trees against tree regular grammars can be considered as execution of tree automata. Automata for regular tree grammars have been studied in the past and present [CDG+97]. 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 elements, while the latter begins with leaf nodes and assigns states to elements after handling subordinate elements. Moreover, there are deterministic tree automata and non-deterministic tree automata: the former assigns a state to each element, while the latter assigns any number of states to each element. As a result, there are four types of tree automata:

- deterministic top-down,
- non-deterministic top-down,
- deterministic bottom-up, and
- non-deterministic bottom-up.

It is known that non-deterministic top-down, deterministic bottom-up and non-deterministic bottom-up tree automata are equally expressive. In other words, any regular tree language can be accepted by some non-deterministic top-down automata. The same thing applies to deterministic bottom-up and non-deterministic bottom-up tree automata. On the other hand, deterministic top-down tree automata are not equally expressive. In other words, some regular tree language cannot be accepted by any deterministic top-down tree automaton.

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. Hence, we call them semi-deterministic top-down.

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.

Other than deterministic tree automata, we have non-deterministic ones and our algorithms in the next subsection are based on them.

4.4 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. Unlike Algorithms 1, 2, and 3, this algorithm does not create string automata from content models.

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 → a r'` such that `a` is the terminal 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 $k\ (1 \le k \le i)$, 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 $k\ (1 \le k \le i)$, 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 algorithm 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.

Observe that this algorithm provides an interpretation of the document, but does *not* ensure that this is the only interpretation. 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 9. Two non-terminals, namely `Para1` and `Para2`, can be assigned to the element `<para/>`. Algorithm 4 merely returns one of the possible interpretations.

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 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 $e_1, e_2, ... e_i$ 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 → 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.

Again, this algorithm does *not* provide a unique interpretation of the document. When more than one interpretation is possible, this algorithm returns all of them.

4.5 Summary of Algorithms and their properties

The following table shows how the various algorithms described in this section compare with respect to the tree automaton they simulate, and the model (event model or tree model) that the algorithm requires.

Algorithms	Class of Tree Automaton	Model the algorithm requires
Algorithms 1, 2, 3	semi- deterministic top-down	Event model
Algorithm 4	non deterministic top-down	Tree model
Not Applicable	deterministic bottom-up	Not applicable
Algorithm 5	non deterministic bottom-up	Event model

5. Evaluating Different XML Schema Language Proposals

In this section, we compare six representative XML schema language proposals: DTD, DSD, XML-Schema, XDuce, RELAX Core, 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.

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 → r`, where `x ∈ N` and `r` is a regular expression over *N*. However, we impose a restriction that all non-terminals described by such production rules can be safely expanded to regular expressions over the other non-terminals. For example, `x → ((y, x, y) | y)` is disallowed. Note that this production rule causes non-regular string languages.

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).

5.1 DTD

DTD as defined in [BPS00] is a local tree grammar. This is enforced by not distinguishing between terminals and non-terminals. Element type declarations of DTDs are production rules, and "element types" of XML 1.0 are terminals as well as non-terminals. Content models are required to be deterministic (see Appendix E of [BPS00]). Attribute-list declarations of DTDs associate attributes to terminals.

As an example, consider a DTD as below:

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

It can be captured by a local tree grammar shown below:

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

5.2 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 (not enough backtracking for the * operator). Therefore, DSD does not accept all regular tree languages. For example, consider a sequence of two `author` elements such that the first matches $Author_1$ only and the second matches $Author_2$ only. The greedy evaluation of DSD tries $Author_1$ for both elements, but does not try $Author_2$ for the second.

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: `P = {book_title → title Expression}`. Here, *Expression* is equivalent to `SomeContentSpecification`.

Context patterns of DSD represent conditions on paths from the root, and may be used in element definitions. Our framework does not directly capture context patterns. However, a recent paper [Mur01] by the first author shows that a pair of a regular tree grammar and a regular path expression for locating nodes can be rewritten as a single regular tree grammar. Such rewriting allows context patterns to be captured in our framework.

5.3 XML-Schema

XML-Schema [TBMM00] represents a single-type tree grammar. Although XML-Schema has many complicated mechanisms, it is not very expressive from the viewpoint of formal language theory. Furthermore, Kawaguchi [Ka01] recently argued that most mechanisms of XML-Schema are confusing and should be avoided.

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*. Most of these features can be described in our framework as illustrated below.

1. A complex type definition 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 → (Title, Author+, Publisher?)`, `Title → title (Pcdata)`, `Author → author (Pcdata)`, and `Publisher → publisher (Pcdata)`.

2. A group definition defines a non-terminal and a production rule without a terminal. An example ([Fal01] Section 2.7) is shown below:

```
<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 production grammar rules `ShipTo → shipTo (USAddress)`, `BillTo → billTo (USAddress)`, and `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 [Fal01] Section 4.2) is given below.

```
<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 above type definitions for Address and UKAddress is equivalent to production rules `Address → (Name, Street, City)`, `UKAddress → (Name, Street, City, Postcode)`, and production rules for `Name, Street, City, Postcode`.

   Now suppose there was an element declaration such as: `<xsd:element name="shipTo" type="Address"/>` This is equivalent to production rules `ShipTo → shipTo (Address)` and `ShipTo → shipTo (UKAddress)`.

   Note that the production 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.

```
P = {X → (Title), Y → (Title, Author1+), Z → (Title, Author2+, Publisher),
     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 ([Fal01] 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:

```
P = {ShipComment → shipComment Y, customerComment → cutomerComment Z
     Ipo:comment → shipComment Y, Ipo:comment → shipComment Z}
```

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

5.4 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 production rule with a terminal. 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 a production rule without a terminal. For example, `type X = T, X | ()` represents a production rule `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 terminals in *T*. For example, consider the subtag declaration

```
subtag i <: fontstyle
```

To convert this subtagging relation into grammar, consider all production 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}`

5.5 RELAX Core

Any regular tree grammar can be expressed in RELAX Core [Mur00b]. The main features of RELAX Core are `elementRule`s and `hedgeRule`s, and clear differentiation between them.

To convert RELAX Core 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 production rule with a terminal. 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 a production rule `Section → section (paraWithFNotes*)`

2. `hedgeRule` defines a production rule without a terminal. For example, consider the `hedgeRule` mentioned in [Mur00b]

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

   The above `hedgeRule` can be converted into `blockElem → (para)`

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

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

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

   This can be converted into production rules `blockElem → (para)` and `blockElem → (itemizedList)`

4. RELAX Core allows multiple `elementRule`s 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 `elementRule`s can be converted into three production rules `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">
  <attribute name="type" required="true">
    <enumeration value="integer"/>
  </attribute>
</tag>
<elementRule role="val-integer" label="Val">
    <ref label="X"/>
</elementRule>
<tag name="val" role="val-string">
  <attribute name="type" required="true">
    <enumeration value="string"/>
  </attribute>
</tag>
<elementRule role="val-string" label="Val">
  <ref label="Y"/>
</elementRule>
```

They can be converted into production rules `Val → val1 X` and `Val → val2 Y`, where `val1` represents `<val type="string">` and `val2` represents `<val type="integer">`.

5.6 TREX

TREX [Cla01] is quite similar to RELAX Core. Major differences of TREX from RELAX Core are (1) content models containing both attributes and elements, (2) interleaving (shuffling) of regular expressions, (3) wild cards for names, and (4) handling of namespaces. Other differences are syntactical. Among these differences, (1) is highly important and is based on new validation techniques. For the lack of space, we do not further consider this issue in this paper.

TREX supports the notion of non-terminals using `define`. However, unlike RELAX Core, TREX does not distinguish between non-terminals that produce trees and non-terminals that produce list of trees. 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:

```
P = {AddressBook → addressBook (Card*), Bold → bold (Inline),
     Italic → italic (Inline), Inline → (Bold | Italic)*}
```

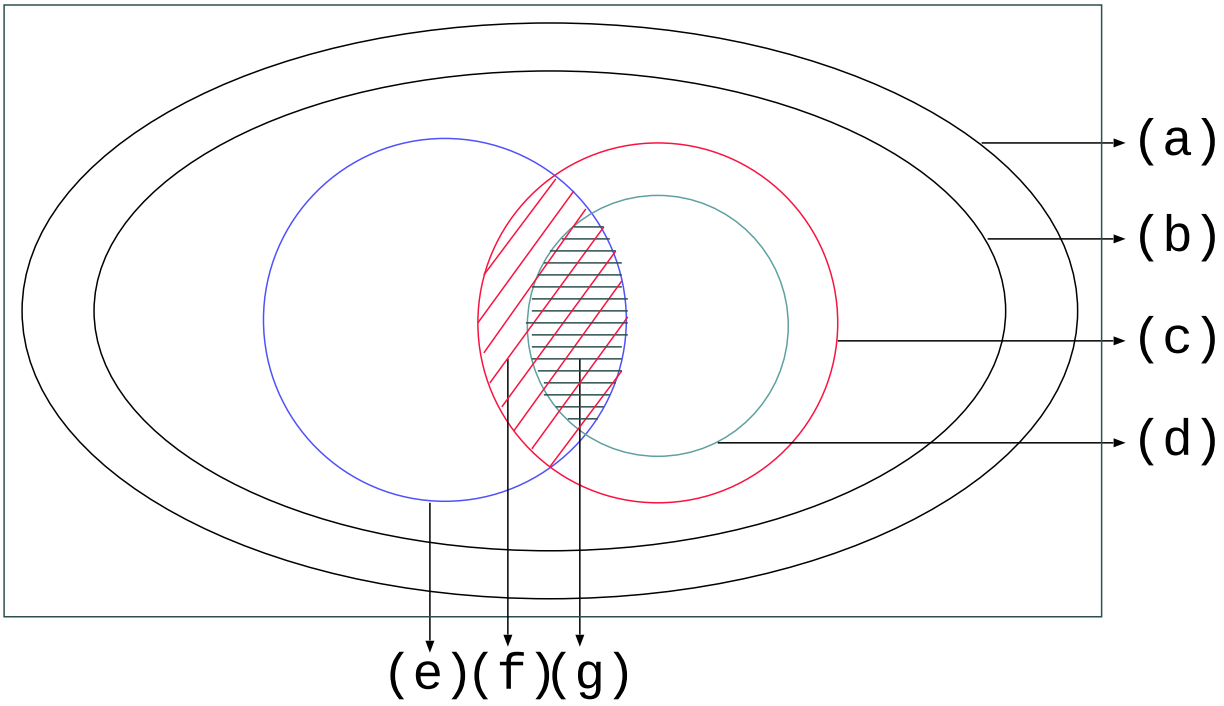

Figure 2.

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

5.7 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 Core:** Four implementations of RELAX Core 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. Its algorithm combines Algorithm 3 (top-down) and Algorithm 5 (bottom-up). When an element is visited, possible non-terminals for this element are computed; unlike Algorithm 3, more than one possible non-terminal may be found. Then, as in Algorithm 5, this program determines which of the possible non-terminals are allowed by the subordinate elements. Advantages of combining Algorithms 3 and 5 are appropriate error messages and error recovery.
    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 can be considered as a combination of Algorithms 3 and 5, but is more advanced; it uses derivatives of regular expressions to construct tree automata lazily. Furthermore, this algorithm efficiently support `<interleave` and attribute-element content models.

6. Conclusion

To compare XML schema language proposals, we have studied four classes of tree languages, namely "local", "single-type", "restrained-competition", and "regular". The class "regular" is the most expressive and is closed under boolean operations, while the other classes are weaker and are not closed under union and difference operations. We have also presented five validation algorithms for these classes. Then, we have shown which class is captured by DTD, DSD, XML-Schema, XDuce, RELAX Core, and TREX, respectively.

Surprisingly enough, from the viewpoint of formal language theory, XML-Schema is not as powerful as RELAX Core or TREX. XML-Schema capture the class "single-type" only, and is not closed under union or difference. On the other hand, RELAX Core and TREX capture the class "regular", which is closed under boolean operations. However, the class "regular" may provide more than one interpretation of a document. One could argue that such multiple interpretations are problematic, and that the class "regular" should be avoided for this reason.

Finally, OASIS recently started RELAX NG [CM01], which unifies RELAX Core and TREX. RELAX NG inherits advantages of both languages, and is likely to become a simple, powerful, and solid language.

Bibliography

[BFRW01] A. Brown, M. Fuchs, J. Robie, and P. Wadler. ``MSL. A model for W3C XML Schema", In 10th Int'l World Wide Web Conf., Hong Kong, May 2001.

[BPS00] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds). ``Extensible Markup Language (XML) 1.0 (2nd Edition)", W3C Recommendation, Oct. 2000. *http://www.w3.org/TR/2000/REC-xml-20001006*.

[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*.

[CM01] J. Clark, and M. Murata (Eds). ``RELAX NG Tutorial", OASIS Working Draft, Jun. 2001. *http://www.oasis-open.org/committees/relax-ng/tutorial.html*.

[Fal01] D. C. Fallside (Eds). ``XML Schema Part 0: Primer", W3C Recommendation, May 2001. *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*.

[Ka01] K. Kawaguchi. ``W3C XML Schema Made Simple", XML.com 2001 *http://www.xml.com/pub/a/2001/06/06/schemasimple.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*.

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

[Mur01] M. Murata. ``Extended Path Expressions for XML", In ACM PODS, Santa Barbara, CA, May 2001.

[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", W3C Recommendation, May 2001. *http://www.w3.org/TR/xmlschema-1/*.

[CCF+01] Don Chamberlin, James Clark, Daniela Florescu, Jonathan Robie, Jérôme Siméon, Mugur Stefanescu (Eds). ``XQuery 1.0: An XML Query Language", W3C Working Draft, June 2001. *http://www.w3.org/TR/xquery/*.

[OA01] OASIS. ``SGML/XML and Forest/Hedge Automata Theory", Web page, May 2001. *http://xml.coverpages.org/hedgeAutomata.html*.

[VV01] V. Vianu. ``A Web Odyssey: From Codd to XML". In ACM PODS, Santa Barbara, CA, May 2001.

[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", W3C Recommendation, Sep. 2000. *http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/*.

Footnotes

1 Throughout the paper, we differentiate two terms -- XML schema(s) and XML-Schema. The former refers to a general term for schema in XML model, while the latter refers to one particular kind of XML schema language proposed by W3C .

Biography

Makoto Murata
IBM Tokyo Research Lab./ International University of Japan
E-mail: mmurata@trl.ibm.co.jp

MURATA Makoto (FAMILY Given) is a researcher working on document processing.

Dongwon Lee
UCLA / CSD
E-mail: dongwon@cs.ucla.edu

Dongwon Lee is a Ph.D candidate majoring in database systems with a focus on XML.

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

Murali Mani is a Ph.D candidate majoring in database systems with a focus on XML. He is partially supported by the NSF grants - 0086116, 0085773, 9817773.