Reasoning about XML Schema Languages using Formal Language Theory

Dongwon Lee
UCLA / CSD
dongwon@cs.ucla.edu

Murali Mani
UCLA / CSD and
IBM Almaden Research Center
mani@cs.ucla.edu

Makoto Murata
IBM Tokyo Research Lab.
mmurata@trl.ibm.co.jp

Technical Report, IBM Almaden Research Center, RJ# 10197, Log# 95071*

November 16, 2000

Abstract

A mathematical framework using formal language theory to describe and compare XML schema languages is presented. Our framework uses the work in two related areas – regular tree languages [CDG$^+$97] and ambiguity in regular expressions [BEGO71, BKW98]. Using these work as well as the content in two classical references [HU79, AU79], we present the following results: (1) a normal form representation for regular tree grammars, (2) a framework of marked regular expressions and model groups, and their ambiguities, (3) five subclasses of regular tree grammars and their corresponding languages to describe XML content models: regular tree languages, TD(1) (top-down input scan with 1-vertical lookahead), single-type constraint languages, TDLL(1) (top-down and left-right input scan with 1-vertical and 1-horizontal lookaheads), and local tree languages, (4) the closure properties of the five language classes under boolean set operations, (5) a classification and comparison of a few XML schema proposals and type systems: DTD, XML-Schema, DSD, XDuce, RELAX, and (6) properties of the grammar classes under two common operations: XML document validity checking and type resolution (i.e., XML document interpretation).

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 regular tree grammar theory, as such a framework. Let us first consider a motivating example.

[*] Available at http://www.cs.ucla.edu/~dongwon/paper/

Example 1: Consider a collection of XML documents, where each document is associated with a different schema. One may want to find an union of two or more schemas (or even union of two "types" defined in one or more schema). Suppose a user wants to obtain a schema that is the union of two schemas. Then, the user would wish that a document that is "valid" with respect to either of the schemas be valid with the union schema, and any document that is "invalid" with respect to both the schemas remain invalid even with respect to the union schema. If a schema evolves into a new version, one may want to take the intersection of the old and the new schemas to determine the documents that still remain valid. Also one may want to take the difference of the new schema from the old schema to determine the semantics and constraints that were supported by the old schema, but not by the new schema. $\Diamond$

Example 1 illustrates the usefulness of the closure properties of XML schema languages under boolean operations (i.e., union, intersection, difference). Such operations are common in real applications, yet they have not been fully investigated in the context of XML schema languages. For instance, heterogeneous XML data integration system extensively uses a union or intersection operator to create a canonical view of underlying sources. In XML query processing, computing answers from multiple documents (or even from a single document) may require to compute union as well. Mathematical framework will help to study these properties of various XML schema languages in a precise way. Let us consider another motivating example.

Example 2: Given an XML document and its schema, suppose one wants to verify if the document is valid against the schema and further find out the types (or non-terminals) associated with each element in the document. Such operation requires the entire document in memory for a regular tree grammar. There is no such requirement for a restricted form of regular tree grammar called $\mathsf{TDLL}(1)$ grammar. $\Diamond$

Example 2 reveals the importance of complexity aspects of some operations performed on XML schema and document. Such issues are directly related with the efficient implementation of XML schema language proposals as well as the popular SAX [Meg00] and DOM [WHA$^+$00] interfaces. We believe it is important to have a mathematical framework to study when an efficient operation is possible and when it is not. This paper is thus our attempt to answer those questions.

Our contributions are, in short, as follows: (1) we propose a mathematical framework using formal language theory; we define ambiguities in marked regular expressions and model groups, (2) we define five subclasses of the regular tree grammars and their corresponding languages to describe XML content models precisely, (3) we study the closure properties of the five subclasses under boolean set operations, (4) 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 (5) we present results on the complexity of two important schema operations – membership checking and type resolution (i.e., XML document interpretation).

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:

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

(a) book.xsd

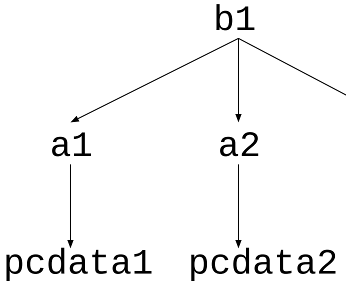
```
<book>
  <author>J. E. Hopcroft</author>
  <author>J. D. Ullman</author>
  <publisher Name="Addison-Wesley"/>
</book>
```

(b) book.xml

Table 1: A DTD and XML document fragment for `book` example.

$$
\begin{aligned}
N &= \{Book, Author, Publisher, Pcdata\}\\
T &= \{book, author, publisher, pcdata\}\\
S &= \{Book\}\\
P&:\\
Book &\rightarrow book(Author^{+}, Publisher)\\
Author &\rightarrow author(Pcdata)\\
Publisher &\rightarrow publisher(\epsilon)\\
Pcdata &\rightarrow pcdata(\epsilon)
\end{aligned}
$$

(a) grammar



(b) instance tree

Table 2: A *grammar* and *instance tree* representation corresponding to XML schema and document in Table 1.

- 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* [TBME00]. Example is shown in Table 1.

- The formal language community (the oldest community to have studied trees as defined in XML) uses terms such as grammars, 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 $\mathcal{T}(V, E)$ (V is the set of nodes and E is the set of edges in $\mathcal{T}$) to denote an instance tree. Example is shown in Table 2.

- The type theorists use 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 considered as being assigned to a *variable* [HVP00]. Example is shown in Table 3.

We use three more terms to describe the information in an XML document: *value-content*[1], *structure* and *element-structure*. An XML document is viewed as value-content embedded within structure. *Value-content* of an XML document refers to text values and attribute values. *Structure* refers to the tree structure and attribute names. *Element-structure* refers to the element tag names arranged as a tree (i.e., structure without attribute names). For defining the element-structure, we replace every text value in the XML document by a node called $\langle pcdata \rangle$. Table 4 shows the example of structure and element-structure. Note that the element-structure of the XML document does not include any text or attribute values.

[1] We define the value-content of an XML document, not to be confused with the content of an element typically used with respect to XML. Also we do not use value-content afterwards, we use it here for the sole purpose of

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

(a) type definitions

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

(b) variables and values

Table 3: A *type definitions* and *variables* representation corresponding to XML schema and document in Table 1.

```
<book>
  <author><pcdata/></author>
  <author><pcdata/></author>
  <publisher Name=/>
</book>
```

(a) structure

```
<book>
  <author><pcdata/></author>
  <author><pcdata/></author>
  <publisher/>
<book/>
```

(b) element-structure

Table 4: A *structure* and *element-structure* representation of the `book.xml` XML document in Table 1.

1.2 Related Work

As mentioned before, our work relies largely on work in two related areas - regular tree languages and ambiguous regular expressions. Tree languages and regular tree languages have a long history dating back to the late 1950's. One of the main reasons for the study in this period was because of their relationship to derivation trees of context-free grammars. A description of tree languages, and their closure properties can be obtained from the book available online [CDG$^+$97]. Our contributions to this field consist of defining a normalized grammar representation for regular tree languages and subclasses of regular tree languages based on the features of most of the XML schema proposals.

A field related to regular tree languages is regular hedge languages studied in [Tak75, Mur00a]. A regular hedge language defines a set of hedges, where a hedge is an ordered list of trees. A regular hedge language is essentially similar to a regular tree language because we need to only add a special root symbol to every hedge in a regular hedge language to make it a regular tree language. In other words, if $L(H)$ is a regular hedge language, then $\{\$\langle w\rangle \mid w \in L(H)\}$ is a regular tree language, where \$ is a special symbol denoting the root of the tree.

Ambiguity in regular expressions is described in [BEGO71]. Here, the authors give several results relevant to our paper: (1) every regular language has a corresponding unambiguous regular expression, (2) we can construct an automaton called Glushkov automaton in [BKW98] corresponding to a given regular expression that preserves the ambiguities, and (3) given a non-deterministic finite state automaton, we can obtain a corresponding regular expression with the same ambiguities.

Ambiguous regular expressions and model groups are studied in the context of SGML content models in [BKW98]. Here the authors describe the concept of fully marked regular expressions, and 1-ambiguous regular expressions and languages. Further, they construct Glushkov automaton that preserves the 1-ambiguity in regular expressions.

Our work includes several extensions to the existing work on ambiguity in regular expressions: (1) we extend the notion of a marked regular expression to allow two or more symbols in a marked regular expression to have the same subscript, (2) we define ambiguities in such marked regular expressions and model groups, (3) we extend Glushkov automaton to our marked regular expressions, and (4) we define prefix and suffix regular expressions or model groups, that can be used to

introducing element-structure of a document which is the topic of this paper.

check whether a given marked expression or model group is ambiguous or 1-ambiguous; this is to be compared against the typical definition of successor sets as the set of symbols that can occur as the successor of a symbol in a regular expression [BEGO71].

1.3 Outline of the paper

The remainder of this paper is organized as follows. In Section 2, we define regular tree languages, regular tree grammars, and NF1, the normalized representation of regular tree grammars. In Section 3, we study regular expressions, marked regular expressions and the ambiguities in marked regular expressions. After this, we introduce subclasses of regular tree languages in Section 4 and study the closure properties of the different language classes under boolean set operations in Section 5. This is followed by an evaluation of the different XML schema language proposals in Section 6, an overview of complexity analysis of document validity checking and document interpretation in Section 7 and then our concluding remarks.

2 Regular Tree Languages and Grammars

Traditional regular string languages or regular string grammars [HU79] are not suitable to describe permissible element content in DTD and other XML schema languages since they are originally designed to describe permissible *strings*, not *element trees* [Mur99a]. Instead these element trees form regular tree languages. In this section, we define regular tree grammars and languages, and some terms that we will use for the remainder of the paper. We borrow some definitions from [CDG$^+$97]. Context-free tree grammars also have been studied in the past [CDG$^+$97], but we restrict ourselves to regular tree grammars.

The definitions of regular tree languages and tree automata are given in [CDG$^+$97]. The book defines a regular tree grammar as follows. (We slightly modify the definition to allow trees with "infinite arity", and allow a model group [BKW98] in the production rules. We also use a notation that we presume more readers are familiar with.)[2]

Definition 1. (Regular Tree Grammar) [CDG$^+$97] A regular tree grammar (RTG) is denoted by a 4-tuple $G = (N, T, P, S)$ where:

- N is the set of non-terminal symbols,
- T is the set of terminal symbols,
- P is the set of production rules of the form "$X \rightarrow a\ Expression$", where $X \in N$, $a \in T$, and $Expression$ is a model group over N.
- S is the set of start symbols, where $S \subseteq N$. □

For instance, the regular tree grammar representation of the `book.xsd` DTD is the same as the grammar representation in Table 2 (a). The above definition is called a normalized regular tree grammar in [CDG$^+$97]. The equivalence of regular tree languages and the grammars as defined in Definition 1 is shown in [CDG$^+$97].

In this context, we would like to mention the difference between a *tree-set* and a *string set*: a regular tree grammar can be said to define either a tree-set or a string set. In other words, we can

[2] We give another equivalent definition for a regular tree grammar when we study the different schema proposals in Section 6.

consider the language defined by a regular tree grammar to define either a set of trees, or a set of serialized strings. Hence, the production rules for a grammar in Table 2 (a) can be re-written as follows.

```
Book -> <book>Author+, Publisher</book>
Author -> <author/>
Publisher -> <publisher/>
```

The reader can observe that there are two orthogonal sub-rules in every production rule of a regular tree grammar. For example, consider a production rule $Book \rightarrow book(Author^{+}, Publisher)$.

- The first sub-rule says that a tree with root symbol *book* can be derived from *Book*.
- The second sub-rule says that such a tree has children as specified by $(Author^{+}, Publisher)$.

Based on these observations, we define a normal form for regular tree grammars in the next subsection.

2.1 Normal Form for Regular Tree Grammars

The normal form definition is intended to give a concise, syntactical representation for the schemas for easier understanding and analysis. Normal forms for regular and context-free string grammars are studied in [HU79]. Regular string grammars have two normal forms: right linear and left linear grammars. Context free grammars also have two normal forms: Chomsky and Greibach normal forms. We define the normal form for regular tree grammars based on the following observations and Lemmas 1 and 2.

- There is an increased interest to separate the two parts in a production rule of a tree-regular grammar in schema proposals. Hedge rules in RELAX [Mur00b], regular expression types in XDuce [HVP00], and complex type definitions in XML-Schema [TBME00] are examples.
- After we separate the two subparts, we should ensure that the content of an element describes a regular string language and not a context-free language. RELAX ensures this by disallowing recursion in hedgeRules, and XDuce allows only right linear grammars.
- Orthogonal restrictions can be placed on the two subparts of a production rule to define subclasses of regular tree languages.

Lemma 1: For a regular tree grammar, if we have two rules of the form "$A \rightarrow a\ X$" and "$A \rightarrow b\ Y$", where $A \in N$, $a, b \in T$, and X and Y are model groups over N, then we can rewrite the regular tree grammar in such a way that for every non-terminal, say $C \in N$, we have only one rule of the form "$C \rightarrow c\ Z$". ■

The proof is by construction: We rewrite $A \rightarrow b\ Y$ as $A1 \rightarrow b\ Y$ and keep $A \rightarrow a\ X$ as it is. Wherever A appears on the RHS of a rule, we replace it by $(A + A1)$.

Lemma 2: For a regular tree grammar, if we have two rules of the form "$A \rightarrow a\ X$" and "$A \rightarrow a\ Y$", then we can rewrite it as "$A \rightarrow a\ (X + Y)$". ■

The proof is that X and Y are model groups, and therefore $X+Y$ is a model group by definition.

Definition 2. (Normal Form 1) A grammar G in *Normal Form 1 (NF1)* is defined as

$$G = (N1, N2, T, P1, P2, S)$$

where

- $N1$ is a set of non-terminal symbols used for deriving trees (We use capitals for the first letter for a symbol in $N1$. e.g., *Book*).
- $N2$ is a set of non-terminal symbols used for content model specification (We use capitals for all letters for a symbol in $N2$. e.g., $BOOK$).
- T is a set of terminal symbols.
- $P1$ is a set of production rules of the form "$A \rightarrow a\ X$", where $A \in N1$, $a \in T$ and $X \in N2$. There is a restriction that there is at most one rule in $P1$ corresponding to a non-terminal symbol in $N1$.
- $P2$ is a set of production rules of the form "$X \rightarrow Expression$", where $X \in N2$, and $Expression$ is a model group over non-terminal symbols in $N1$. There is a restriction that there is at most one rule in $P2$ corresponding to a non-terminal symbol in $N2$.
- S is a set of start symbols, where $S \subseteq N1$. □

We can convert any regular tree grammar in Definition 1 to NF1 based on the constructions described in Lemmas 1 and 2, and then separating the two subparts in every production rule. Sometimes for ease of explanation, we combine a $P1$ rule such as $A \rightarrow a\ X$, with a $P2$ rule such as $X \rightarrow Expression$, and say that the production rule for A is $A \rightarrow a\ Expression$.

Theorem 1: *Every regular tree language can be derived from a regular tree grammar in NF1.* ■

Example 3: Consider the grammar G in Table 2 (a). If we convert G to NF1, then we get $G_3 = (N1, N2, T, P1, P2, S)$, where

$$\begin{aligned}
N1 &= \{Book, Author, Publisher, Pcdata\} \\
N2 &= \{BOOK, AUTHOR, PUBLISHER, PCDATA\} \\
T &= \{book, author, publisher, pcdata\} \\
P1 &= \{Book \rightarrow book\ BOOK, Author \rightarrow author\ AUTHOR, \\
&\quad Publisher \rightarrow publisher\ PUBLISHER, Pcdata \rightarrow pcdata\ PCDATA\} \\
P2 &= \{BOOK \rightarrow (Author^{+}, Publisher), \\
&\quad AUTHOR \rightarrow Pcdata, PUBLISHER \rightarrow \epsilon, PCDATA \rightarrow \epsilon\} \\
S &= \{Book\}
\end{aligned}$$

◊

For a regular tree grammar, G in NF1, and an instance tree $\mathcal{T}(V, E) \in L(G)$ as given in Table 2 (b) we define the following.

- $root(A), A \in N1$: This defines the root of the tree produced by A. In other words, $root(A) = t$, if there is a production rule in $P1$ such as $A \rightarrow t\ X, X \in N2$. For a regular tree grammar in NF1, $root(A)$ is unique for any $A \in N1$.

- $unmarkedContentModel(A), A \in N1$: This is the model group over T denoting the content of A. For instance, $unmarkedContentModel(Book) = (author^{+}, publisher)$.

- $contentModel(A), A \in N1$: This denotes the model group over $N1$ denoting the content of A. For instance, $contentModel(Book) = (Author^{+}, Publisher)$.

- $assignmentTree(\mathcal{T}(V, E))$: The assignment tree gives the tree of non-terminal symbols in $N1$ denoting a valid assignment for $\mathcal{T}$. In an assignment tree, there is one non-terminal assignment corresponding to each $v \in V$. An instance tree can have multiple assignment trees for a given regular tree grammar. For the grammar in Example 3, the tree in Table 2 (b) has only one assignment tree, and the assignment for the node $b1$ in this assignment tree is $\{Book\}$.

We can define all these terms (with slight variations) for a regular tree grammar as given in Definition 1. Because all the interesting properties of a regular tree grammar are maintained during normalization, and for convenience, we define the above terms for a regular tree grammar in NF1.

In Section 4, we study the subclasses of regular tree languages, but before that we introduce ambiguities in string regular expressions in the next section.

3 Ambiguities in Regular String Expressions

The content model of a non-terminal symbol $A \in N1$ as defined in the previous section is given by a *model group*. In the next section, we will describe properties of regular tree languages based on the ambiguities in the content models. As a preparation, we study ambiguities in model groups in this section.

We borrow and extend some definitions from [BEGO71]. We also use definitions from [BKW98], where the authors study 1-ambiguous model groups and languages. For the remainder of this section, we will use the model group $F = (a^{*}, b?, a^{*})$ as an illustrative example. We give the following basic definitions for a model group. (Since all the definitions, theorems and lemmas for model groups apply to regular expressions as well, we do not explicitly consider regular expressions.)

- *Alphabet* is the set of symbols used in the model group. For example, $Alphabet(F) = \{a, b\}$.

- *Letter* is a term in the model group [BEGO71]. Further, the ordered list of *letters* in a model group, E, is denoted as *Letters(E)*. For example, $Letters(F) = \langle x, y, z \rangle$. For a model group E, each letter, $l \in Letters(E)$[3] has a corresponding symbol $a \in Alphabet(E)$, we denote this as $sym(l) = a$. In the above example, $sym(x) = a, sym(y) = b, sym(z) = a$. The relationship $|Letters(E)| \geq |Alphabet(E)|$ holds for every model group.

- For a symbol a_i, we call a the *base*, and i its *subscript*.

- *Marking* [BKW98] is a process of adding a subscript to each letter in a model group. We denote marking of a model group by m, for example, we can define $m(F) = (a_1^{*}, b_1?, a_1^{*})$. The result of marking a model group is a *marked model group* defined below.

- A *Marked Model Group* is a model group that consists of only letters with subscripts [BKW98]. We typically use E', F', and G' to denote a marked model group. For example, $F' = (a_1^{*}, b_1?, a_1^{*})$ and $G' = (a_1^{*}, b_1?, a_2^{*})$ are both marked model groups for E. Since any marked

[3] We use $\in$ to denote element membership for a list also.

model group E' is also a model group, we can define $\mathit{Letters}(E')$ and $\mathit{Alphabet}(E')$ accordingly, where $|\mathit{Letters}(E')| \geq |\mathit{Alphabet}(E')|$.

A *marked regular expression* is defined similar to a marked model group. (Note that our marked model group (or regular expression) is an extension of the marked regular expression defined in [BKW98], where they require that no two letters in the marked regular expression be same.)

- *Unmarking* is the inverse of marking. In unmarking, we drop subscripts from a marked model group (or regular expression). The result of unmarking a marked model group E' is a model group E. We denote the unmarking function by h. For example, $h(F') = h(G') = (a^*, b?, a^*)$ [BKW98]. We also define $h(w')$ where $w' \in L(E')$, as well as $h(a')$, where $a' \in \mathit{Alphabet}(E')$. For example, $h(a_1, a_1, a_2) = (a, a, a)$, and $h(a_1) = a$.

- Two symbols, $x, y \in \mathit{Alphabet}(E')$ are called *competing symbols* if $x \neq y$, and $h(x) = h(y)$. In other words x and y have the same base but different subscript. For example, the two symbols a_1 and a_2 in G' are competing, whereas there are no competing symbols in F'.

- Two letters, $l, m \in \mathit{Letters}(E')$ are called *competing letters* if $\mathit{sym}(l) = x, \mathit{sym}(m) = y$, and x and y are competing symbols. For example, the first and third letters in E' are competing letters, whereas there are no competing letters in F'.

- A *Fully Marked Model Group* is a marked model group that has no two letters l, m such that $\mathit{sym}(l) = \mathit{sym}(m)$. For example, $F'' = G' = (a_1^*, b_1?, a_2^*)$ is a fully marked model group, whereas $F' = (a_1^*, b_1?, a_1^*)$ is not. We typically use E'', F'', and G'' to denote fully marked model groups. For a fully marked model group E'', $|\mathit{Letters}(E'')| = |\mathit{Alphabet}(E'')|$.

  A fully marked regular expression is defined similar to a fully marked model group. (Note that the marked regular expression as defined in [BKW98] corresponds to our fully marked regular expressions.)

- We use fm to denote conversion of a model group or a marked model group to a fully marked model group. From our previous examples, $fm(F) = fm(F') = fm(G') = fm(F'') = F''$.

- We use h' to denote conversion a fully marked model group to a marked model group. For example, $h'(F'')$ can be defined as either F' or G'. Like unmarking, we define h' for $w \in L(E'')$ as well as for $a'' \in \mathit{Alphabet}(E'')$. For example, $h'(a_1, a_1, a_2)$ can be (a_1, a_1, a_1) or (a_1, a_1, a_2)[4] Similarly $h'(a_2)$ can be either a_1 or a_2.

  Note that if two letters $x, y \in \mathit{Letters}(E')$ are competing, then the corresponding letters in $\mathit{Letters}(E'')$ must be competing, where $fm(E') = E''$.

- We denote a model group (marked or unmarked) over the alphabet Σ as $MG(\Sigma)$. Similarly we use $RE(\Sigma)$ to denote a regular expression (marked or unmarked) over Σ.

The use of subscripted symbols is only for easier explanation, and holds no significance whatsoever. We can define a function h for any model group E, and obtain a model group $F = h(E)$. In our framework, E corresponds to a marked model group, and F corresponds to the unmarked model group.

[4] Remember that (a_1, a_1, a_2) is a string in the fully marked regular expression $(a_1^*, b_1?, a_2^*)$.

We give two more definitions - *prefix model group* and *suffix model group* before we define ambiguities in model groups. These definitions illustrate how we can obtain a prefix or suffix model group given a model group, E, and a letter, $a \in \mathit{Letters}(E)$.[5]

Definition 3. (Prefix Model Group) For any marked or unmarked model group E and a letter $a \in \mathit{Letters}(E)$, the prefix model group of a, denoted by $\mathit{prefixMG}(E, a)$, is defined constructively as follows.

$$
\begin{array}{llll}
If & E = a, & then & prefixMG(E,a) = \epsilon \\
If & E = (X), & then & prefixMG(E,a) = prefixMG(X,a) \\
If & E = (X)?, & then & prefixMG(E,a) = prefixMG(X,a) \\
If & E = (X)^*, & then & prefixMG(E,a) = X^*, prefixMG(X,a) \\
If & E = (X)^+, & then & prefixMG(E,a) = X^*, prefixMG(X,a) \\
If & E = (X,Y), & then & prefixMG(E,a) = \left\{ \begin{array}{ll} (X, prefixMG(Y,a)) & if\ a \in \mathit{Letters}(Y) \\ prefixMG(X,a) & if\ a \in \mathit{Letters}(X) \end{array} \right. \\
If & E = (X+Y), & then & prefixMG(E,a) = \left\{ \begin{array}{ll} prefixMG(Y,a) & if\ a \in \mathit{Letters}(Y) \\ prefixMG(X,a) & if\ a \in \mathit{Letters}(X) \end{array} \right. \\
If & E = (X\&Y), & then & prefixMG(E,a) = \left\{ \begin{array}{ll} (X?, prefixMG(Y,a)) & if\ a \in \mathit{Letters}(Y) \\ (Y?, prefixMG(X,a)) & if\ a \in \mathit{Letters}(X) \end{array} \right.
\end{array}
$$

□

Definition 4. (Suffix Model Group) For any marked or unmarked model group E and a letter $a \in \mathit{Letters}(E)$, the suffix model group of a, denoted by $\mathit{suffixMG}(E, a)$, is defined constructively as follows.

$$
\begin{array}{llll}
If & E = a, & then & suffixMG(E,a) = \epsilon \\
If & E = (X), & then & suffixMG(E,a) = suffixMG(X,a) \\
If & E = (X)?, & then & suffixMG(E,a) = suffixMG(X,a) \\
If & E = (X)^*, & then & suffixMG(E,a) = suffixMG(X,a), X^* \\
If & E = (X)^+, & then & suffixMG(E,a) = suffixMG(X,a), X^* \\
If & E = (X,Y), & then & suffixMG(E,a) = \left\{ \begin{array}{ll} suffixMG(Y,a)) & if\ a \in \mathit{Letters}(Y) \\ (suffixMG(X,a), Y) & if\ a \in \mathit{Letters}(X) \end{array} \right. \\
If & E = (X+Y), & then & suffixMG(E,a) = \left\{ \begin{array}{ll} suffixMG(Y,a) & if\ a \in \mathit{Letters}(Y) \\ suffixMG(X,a) & if\ a \in \mathit{Letters}(X) \end{array} \right. \\
If & E = (X\&Y), & then & suffixMG(E,a) = \left\{ \begin{array}{ll} (suffixMG(Y,a), X?) & if\ a \in \mathit{Letters}(Y) \\ (suffixMG(X,a), Y?) & if\ a \in \mathit{Letters}(X) \end{array} \right.
\end{array}
$$

□

Now we are ready to define ambiguous marked model groups as well as 1-ambiguous marked model groups. In our framework, the ambiguous and 1-ambiguous regular expression definitions in [BEGO71, BKW98] are translated to ambiguous and 1-ambiguous fully marked regular expressions.

Definition 5. (Ambiguous Marked Model Group) [BEGO71] A marked model group E' is ambiguous if $L(E')$ contains two words uxv and $u'yv'$, where u, u', v, v' are words over $\mathit{Alphabet}(E')$, and $x, y \in \mathit{Alphabet}(E')$, such that $h(u) = h(u')$, $h(v) = h(v')$, $x \neq y$ and $h(x) = h(y)$. □

[5] Defining prefix and suffix model groups as opposed to the much easier prefix and suffix languages based on the well-established notions of left and right quotients is purely a matter of taste and judgement.

Example 4: The marked model group $G' = (a_1^*, b_1?, a_2^*)$ is ambiguous. For example, consider $u = u' = a_1^m, v = v' = a_2^n$, and $x = a_1, y = a_2$. ◊

Definition 6. (1-Ambiguous Marked Model Group) [BKW98] A marked model group E' is 1-ambiguous if $L(E')$ contains two words uxv and $u'yw$, where u, u', v, w are words over $Alphabet(E')$, and $x, y \in Alphabet(E')$, such that $h(u) = h(u')$, $x \neq y$ and $h(x) = h(y)$ [6]. □

Example 5: The marked model group $G' = (a_1^*, b_1?, a_2^*)$ is 1-ambiguous, if you consider $u = u' = a_1^m, v = a_1^n, w = a_2^p$ and $x = a_1, y = a_2$. On the other hand, $E' = (a_1^*, b_1?, a_2)$ is a 1-ambiguous marked regular expression, but not ambiguous. ◊

We can define ambiguous and 1-ambiguous marked regular expressions similar to ambiguous and 1-ambiguous model group. Now we show how we can determine whether a given model group (or regular expression) is ambiguous or 1-ambiguous.

Theorem 2: *A marked model group E' is ambiguous if and only if there exist two competing letters, $l, m \in Letters(E')$, such that*

- $L\ (h\ (prefixMG\ (E', l))) \cap L\ (h\ (prefixMG\ (E', m))) \neq \phi$, *and*
- $L\ (h\ (suffixMG\ (E', l))) \cap L\ (h\ (suffixMG\ (E', m))) \neq \phi$.

■

Proof: We prove both directions.

"→ direction": From the definition, there exist two strings $uxv, u'yv' \in L(E')$. If $x \neq y$, and $h(x) = h(y)$, then x and y are competing symbols in $Alphabet(E')$. Let l and m be two letters in E' that correspond to x and y for the strings uxv and $u'yv'$. Because x and y are competing symbols, $l \neq m$. Further, we have $h(u) = h(u') \in L\ (h\ (prefixMG\ (E', l))) \cap L\ (h\ (prefixMG\ (E', m)))$, and $h(v) = h(v') \in L\ (h\ (suffixMG\ (E', m))) \cap L\ (h\ (suffixMG\ (E', m)))$.

"← direction": Let s_1 be a string in $L\ (h\ (prefixMG\ (E', l))) \cap L\ (h\ (prefixMG\ (E', m)))$, and s_2 be a string in $L\ (h\ (suffixMG\ (E', l))) \cap L\ (h\ (suffixMG\ (E', m)))$. Let u be a string in $L\ (prefixMG\ (E', l))$, and u' be a string in $L\ (prefixMG\ (E', m))$, such that $h(u) = h(u') = s_1$. Similarly let v be a string in $L\ (suffixMG\ (E', l))$, and v' be a string in $L\ (suffixMG\ (E', m))$, such that $h(v) = h(v') = s_2$. The existence of u, u', v, v' follow from the definitions. Let $x = sym(l)$ and $y = sym(m)$, now $uxv, u'yv' \in L(E')$. ◆

Theorem 3: *A marked model group E' is 1-ambiguous if and only if there exist two competing letters, $l, m \in Letters(E')$, such that*

- $L\ (h\ (prefixMG\ (E', l))) \cap L\ (h\ (prefixMG\ (E', m))) \neq \phi$

■

Proof: We prove both directions.

"→ direction": From the definition, there exist two strings $uxv, u'yw \in L(E')$. As before, we know that x and y are competing symbols in $Alphabet(E')$, also let $l, m \in Letters(E')$ correspond to x and y respectively for the strings uxv and $u'yw$. We also have $h(u) = h(u') \in L\ (h\ (prefixMG\ (E', l))) \cap$

[6] Actually the definition in [BKW98] requires $u = u'$. It is easy to show that the two definitions are equivalent.

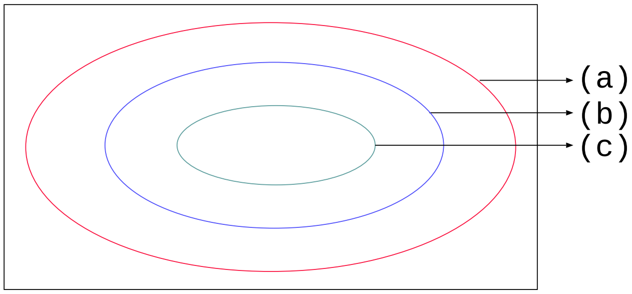


Figure 1: The relationships among different model groups: (a) marked model group, (b) unambiguous marked model group, and (c) 1-unambiguous marked model group.

$L\ (h\ (prefixMG\ (E', m)))$.

"$\leftarrow$ direction": Let s_1 be a string in $L\ (h\ (prefixMG\ (E', l)))\ \cap L\ (h\ (prefixMG\ (E', m)))$. Let u be a string in $L\ (prefixMG\ (E', l))$, and u' be a string in $L\ (prefixMG\ (E', m))$ such that $h(u) = h(u') = s_1$. Let v be any string in $L\ (suffixMG\ (E', l))$, and w be any string in $L\ (suffixMG\ (E', m))$ From the definitions u, u', v, w exist. Let $x = sym(l)$ and $y = sym(m)$, now $uxv, uyw \in L(E')$. ⧫

We have so far defined ambiguous and 1-ambiguous marked model groups and regular expressions. Figure 1 illustrates the relationships among different model groups. The following theorem shows that the ambiguities in marked model groups can be defined using ambiguities in marked regular expressions.

Theorem 4: *A marked or fully marked model group is ambiguous or 1-ambiguous if and only if the "corresponding" marked regular expression is ambiguous or 1-ambiguous.* ■

Proof: The "corresponding" regular expression (marked or unmarked) for a model group (marked or unmarked) is obtained by writing the model group using the regular expression operators as described in [BKW98]. Showing that the ambiguities are preserved is straightforward. ⧫

The reader should observe that the conversion mentioned in Theorem 4 can produce a non fully marked regular expression corresponding to a fully marked model group. Actually there might not exist a fully marked regular expression with the same 1-ambiguity. For example, consider the example from [BKW98], where $E' = (a_1?\ \&\ b_1)$. This translates to $F' = (a_1, b_1 + b_1 + b_1, a_1)$, which does not have a corresponding 1-ambiguous fully marked regular expression.

In the next section, we will see how the ambiguities in model groups relate to regular tree grammars. The astute reader will be able to predict the relationship - for a symbol $A \in N1$, $unmarkedContentModel(A) = h(contentModel(A))$.

4 Subclasses of Regular Tree Grammars

In this section, we first show the relationship between the marked model groups which we studied in the last section and regular tree grammars. After this we define subclasses of regular tree languages which can be defined based on ambiguities in marked model groups. At the end of this section, we show the relationship between the various language classes.

4.1 Regular Tree Grammars and Ambiguous Regular Expressions

The relationship between regular tree grammars and ambiguous model groups is because the content model of a non-terminal symbol is a model group. We will consider a regular tree grammar in NF1, for convenience. The reader will be able to observe by the end of this section that the transformation of a regular tree grammar as given in Definition 1 to a regular tree grammar in NF1 preserves ambiguities. We again start with a list of basic definitions for a regular tree grammar $G = (N1, N2, T, P1, P2, S)$.

- *TreeLanguage*: We define *TreeLanguage* of a symbol $A \in N1$, denoted $TL(A)$ as the language obtained from $G' = (N1, N2, T, P1, P2, \{A\})$, i.e., the language obtained when $S = \{A\}$.

- We define an unmarked model group corresponding to $\mathit{contentModel}(C)$, where $C \in N1$ as the unmarked content model of C. For example, the unmarked model group corresponding to $\mathit{contentModel}(\mathit{Book})$ is $(\mathit{author}^+, \mathit{publisher})$.

- We say that two symbols $A, B \in N1$ are *competing* if $A \neq B$, and $TL(A) \cap TL(B) \neq \phi$.

  For three symbols $A, B, C \in N1$, if A competes with B and A competes with C, then B need not compete with C. For example, consider $A, B, C \in N1$, and $t, a, d \in T$ such that $TL(A) = t\langle a^*, d^*\rangle$, $TL(B) = t\langle a^+\rangle$ and $TL(C) = t\langle d^+\rangle$. Now A, B compete and A, C compete, but B, C do not.

- We say that two symbols $A, B \in N1$ are *1-lookahead competing* if $A \neq B$ and $\mathit{root}(A) = \mathit{root}(B)$. (We will use competing symbols to denote 1-lookahead competing symbols.)

  It is easy to observe the following: for three symbols $A, B, C \in N1$, if A, B are 1-lookahead competing, and A, C are 1-lookahead competing, then B, C necessarily need to be 1-lookahead competing.

- We define a marked model group that corresponds to the 1-lookahead competing relationship for a content model as follows: Let $MG = \mathit{contentModel}(C)$, where $C \in N1$. For any two 1-lookahead competing symbols in MG, introduce two competing symbols for the marked model group. For example, a marked model group corresponding to $\mathit{contentModel}(\mathit{Book})$ is $E' = (\mathit{author}_1^+, \mathit{publisher}_1)$.

  We often call the marked model group corresponding to the 1-lookahead competing relationship for C as simply the marked model group for C.

- We define a fully marked model group corresponding to any $MG = \mathit{contentModel}(C)$ as any fully marked model group corresponding to $\mathit{unmarkedContentModel}(C)$. For example, E' is also a fully marked model group corresponding to $\mathit{contentModel}(\mathit{Book})$.

With this introduction, we are ready to define subclasses of regular tree languages. We will assume without loss of generality that the regular tree grammar is in NF1. We show that this assumption is without loss of generality towards the end of this section.

4.2 Local Tree Languages and Grammars

Local tree grammars impose restrictions on the rules of P1. Local tree languages and grammars have also been studied in the past [Tak75, Mur99a, Mur99b]. Another commonly used term for *tree-locality constraint* in recent times is *context independent types*.

Definition 7. (Tree-Locality Constraint) *Tree-locality constraint* imposes the restriction on the production rules $P1$ of a regular tree grammar: there is at most one rule in $P1$ of the form "$A \rightarrow a\ X$" for every terminal symbol $a \in T$. □

This constraint ensures the following. Suppose, we have a regular tree grammar, say $G = (N1, N2, T, P1, P2, S)$, that satisfies the tree-locality constraint. Let $\mathcal{T} \in L(G)$ be a tree. Now, given a terminal symbol $a \in T$ (i.e., a is a node in $\mathcal{T}$), there is exactly one non-terminal symbol, $A \in N1$, that has a production rule in $P1$ of the form "$A \rightarrow a\ X$, where $X \in N2$", (i.e., there is only one rule in $P1$ that has a in the RHS).

Definition 8. (Local Tree Grammar) A *local tree grammar (LTG)* is a regular tree grammar that satisfies the tree-locality constraint. □

Example 6: The following grammar $G_6 = (N1, N2, T, P1, P2, S)$ is a local tree grammar.

$$\begin{aligned}
N1 &= \{Book, Author1, Son, Pcdata\} \\
N2 &= \{BOOK, AUTHOR1, SON, PCDATA\} \\
T &= \{book, author, son, pcdata\} \\
S &= \{Book\} \\
P1 &= \{Book \rightarrow book\ BOOK, Author1 \rightarrow author\ AUTHOR1, Son \rightarrow son\ SON, \\
&\quad Pcdata \rightarrow pcdata\ PCDATA\} \\
P2 &= \{BOOK \rightarrow (Author1), AUTHOR1 \rightarrow (Son), SON \rightarrow Pcdata, PCDATA \rightarrow \epsilon\}
\end{aligned}$$

◊

Theorem 5: *The marked model group corresponding to $MG = contentModel(C)$, for any $C \in N1$ in a local-tree grammar is unambiguous and 1-unambiguous.* ■

Proof: From the definition it follows that there are no competing symbols in the marked model group corresponding to MG. ⧫

Definition 9. (Local Tree Language) A language is a local tree language if and only if it can be generated by a local tree grammar. □

4.3 TD(1) Languages and Grammars

TD(1) languages correspond to languages that are deterministic with 1-vertical lookahead of the input tree. In the last section, we introduced the marked model group corresponding to 1-vertical lookahead. We use that for defining TD(1) grammars.

Definition 10. (1-vertical Lookahead Constraint) The *1-vertical lookahead constraint* imposes the following condition on the P2 rules: the marked model group corresponding to $contentModel(C)$, for any $C \in N1$ is not ambiguous. □

Definition 11. (TD(1) Grammar) A TD(1) *grammar* is a regular tree grammar that satisfies the 1-vertical lookahead constraint. □

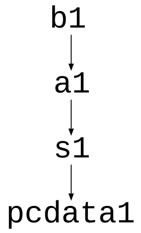
Theorem 6: *If G is a* TD(1) *grammar, and $\mathcal{T} \in L(G)$, then there is a unique assignment tree for $\mathcal{T}$.* ■

```
<!DOCTYPE book [
  <!ELEMENT book    (author)>
  <!ELEMENT author  (son)>
  <!ATTLIST author  Name CDATA>
  <!ELEMENT son     (#PCDATA)>
]>
```

(a) `book2.xsd`

```
<book>
  <author Name="A">
    <son> aa </son>
  </author>
</book>
```

(b) `book2.xml`



(c) instance tree $\mathcal{T}_6$

Table 5: A DTD, XML file and instance tree for `book2`.

Example 7: Consider the local tree grammar G_6 in Example 6 again. Further, suppose G_6 is a grammar representation of `book2.xsd` in Table 5. The example XML document `book2.xml` and its instance tree are also shown in Table 5. There is a single assignment tree corresponding to $\mathcal{T}_6$, and the assignments for the various nodes in this assignment tree are as follows: $assignment(b1) = \{Book\}$, $assignment(a1) = \{Author\}$, $assignment(s1) = \{Son\}$, and $assignment(pcdata1) = \{Pcdata\}$. ◊

Definition 12. (TD(1) language) A language is a TD(1) language if and only if it can be generated by a TD(1) grammar. □

4.4 Single-Type Constraint Languages and Grammars

We introduce the single-type constraint which is a stricter condition than the 1-vertical lookahead constraint.

Definition 13. (Single-Type Constraint) The *single-type constraint* imposes the following restriction on the rules in P2: there are no competing symbols in the marked model group corresponding to $contentModel(C)$, for any $C \in N1$. □

Definition 14. (Single-Type Constraint Grammar) A regular tree grammar is said to be a single-type constraint grammar if and only if it satisfies the single-type constraint. □

Example 8: Consider a regular tree grammar $G_8 = (N1, N2, T, P1, P2, S)$, where

$$\begin{aligned}
N1 &= \{Book, Author1, Son, Article, Author2, Daughter\}\\
N2 &= \{BOOK, AUTHOR1, SON, ARTICLE, AUTHOR2, DAUGHTER\}\\
T &= \{book, author, son, daughter\}\\
S &= \{Book, Article\}\\
P1 &= \{Book \to book\ BOOK, Author1 \to author\ AUTHOR1, Son \to son\ SON,\\
&\quad Article \to article\ ARTICLE, Author2 \to author\ AUTHOR2,\\
&\quad Daughter \to daughter\ DAUGHTER\}\\
P2 &= \{BOOK \to (Author1), AUTHOR1 \to (Son), SON \to \epsilon,\\
&\quad ARTICLE \to (Author2), AUTHOR2 \to (Daughter), DAUGHTER \to \epsilon\}
\end{aligned}$$

G_8 satisfies the single-type constraint since all rules in $P2$ have single symbol in their RHS. However, G_8 does not satisfy the tree-locality constraint since there are two rules in $P1$, "$Author1 \to$

*author AUTHOR*1" and "*Author*2 → *author AUTHOR*2" that produce the same terminal symbol *author*. ◊

Definition 15. (Single-Type Constraint Language) A language is a single-type constraint language if and only if it can be generated by a single-type constraint grammar. □

4.5 TDLL(1) Languages and Grammars

Because trees have vertical and horizontal navigation, we can impose restrictions on the horizontal lookahead also. This is based on the 1-ambiguity definitions. In this paper, we restrict ourselves to deterministic content model definition as in XML 1.0. In other words we study 1-ambiguity in fully marked model groups as in [BKW98]. However, one need not be restricted to deterministic content models and one can define 1-ambiguity for general marked model groups also.[7]

Definition 16. (TDLL(1) Grammar) A TDLL(1) grammar is a regular tree grammar that satisfies the following condition: the fully marked model group corresponding to a $contentModel(C)$, where C is any symbol in $N1$ is 1-unambiguous. □

Example 9: Consider a grammar G_9 that has a rule in $P2$ of the form "$X \rightarrow (A^*, B^*, A, D)$" and rules in $P1$ of the form "$A \rightarrow a\ L$", "$B \rightarrow b\ M$", and "$D \rightarrow d\ N$". G_9 is not TDLL(1) as defined in Definition 16 (The astute reader might observe that the P2 rule is 1-unambiguous if we do not enforce the deterministic constraint). ◊

Definition 17. (TDLL(1) Language) A language is a TDLL(1) language if and only if it can be generated by a TDLL(1) grammar. □

4.6 Relationship between the Language Classes

In this subsection, we examine the relationship between the various language classes we introduced earlier in this section. These relationships are shown in Figure 3 in Section 6, where we compare the expressive power of the different XML Schema proposals. But first, we show that our assumption that the regular tree grammar is in NF1 is without loss of generality.

Lemma 3: The assumption that the regular tree grammar is in NF1 is without loss of generality. ■

Proof: We prove this by defining content model for a regular tree grammar as in Definition 1. Let $G = (N, T, P, S)$ be a regular tree grammar as in Definition 1. Consider $A \in N, t \in T$, such that all the rules that produce t as the root symbol from A be $A \rightarrow t\ E_1, A \rightarrow t\ E_2, \ldots, A \rightarrow t\ E_n$, where $E_1, E_2, \ldots, E_n$ are model groups over N. Now for such a regular tree grammar, we would define $contentModel(A, t) = (E_1 + E_2 + \ldots + E_n)$. We define ambiguities using such content model definitions. It is easy to verify based on our normalization steps in Lemmas 1 and 2 that the ambiguities are preserved in the normalized regular tree grammar. ♦

Now we show the relationship between the various language classes we studied. The proof for these relationships is based on the definition of a non-redundant regular tree grammar. A non-

[7] One reason for defining TDLL(1) grammars only for deterministic content models is that we have not come across any schema proposal which does not use the deterministic content model. This is surprising given that most schema authors are unhappy with this constraint. Also we find TDLL(1) grammars without the deterministic constraint to be a very useful class of grammars, refer Section 8.

redundant regular tree grammar is a regular tree grammar in NF1 which is reduced [CDG$^+$97], that has no two symbols, $A, B \in N1$ such that $TL(A) = TL(B)$, and for which the marked model group corresponding to $contentModel(C)$, where C is any symbol in $N1$ is a *reduced marked model group*.

We first define a reduced marked model group. A reduced marked model group is a marked model group which cannot be rewritten with lesser ambiguities in XML. Now a marked model group in XML corresponds to $contentModel(A)$, for a symbol $A \in N1$. Therefore combining letters is slightly different, for example, you can rewrite $(a_1 + a_2)$ as (a_3), but you cannot rewrite (a_1^*, a_2^*). The scenarios when the letters in a marked model group can be rewritten in XML with fewer ambiguities are as follows:

- We have a + or equivalent operator (?, &). For example,
  - The marked model group $E' = (a_1 + a_2)$ can be rewritten as $F' = (a_3)$. Now E' is ambiguous, but F' is not.
  - The marked model group $E' = (a_1^*, b_1?, a_1^*)$ can be written as $F' = (a_1^* + a_1^*, b_1, a_1^*)$. Now the fully marked regular expression corresponding to E' is ambiguous, and the one corresponding to F' is not.
  - The marked model group $E' = (a_1 \& a_1)$ can be written as $F' = (a_1, a_1)$. The fully marked model group corresponding to E' is ambiguous, and the one corresponding to F' is not.
- We have parantheses in the marked model group. For example, the marked model group $E' = ((a_1^*, a_2^*)^*)$ can be rewritten as $((a_1 + a_2)^*)$, which can be rewritten as $F' = (a_3^*)$. Now E' is ambiguous, but F' is not.
- We have two consecutive letters $x, y \in Letters(E')$, such that $sym(x) = sym(y)$. For example, the marked model group $E' = (a_1, a_1^*)$ can be rewritten as $F' = (a_1^+)$. Now the fully marked model group corresponding to E' is 1-ambiguous, but the one corresponding to F' is not.

Definition 18. (Reduced Marked Model Group) A *reduced marked model group*, E', is a marked model group if it has the following properties.

- E' has no parantheses.
- E' does not have the operators +, ? or &.
- There are no two consecutive letters, $x, y \in Letters(E')$, such that $sym(x) = sym(y)$.

□

Definition 19. (Non-redundant Regular Tree Grammar) A *non-redundant regular tree grammar* is a regular tree grammar $G = (N1, N2, T, P1, P2, S)$ which satisfies the following conditions.

1. There is no symbol in $N1 \cup N2$ that cannot be reached from a symbol in S.
2. There is no symbol $A \in N1$ such that $TL(A) = \phi$ or ϵ.
3. There are no two symbols $A, B \in N1$ such that $TL(A) = TL(B)$.

4. The marked model group corresponding to *contentModel*(A), for any $A \in N1$ is a reduced marked model group.

□

Lemma 4: For any regular tree language that has a non-redundant regular tree grammar representation, the ambiguities for the language are those given by this non-redundant grammar representation. ■

Proof: The proof is that any other regular tree grammar representation will have the same ambiguities in the non-redundant grammar representation. Based on properties 1-3, we cannot reduce the number of symbols in $N1 \cup N2$ without changing the language. Property 4 and the definition of a reduced marked model group ensures that we cannot rewrite a model group so that the resulting model group will have less ambiguities. ♦

Theorem 7: TD(1) *languages is properly contained in regular tree languages.* ■

Proof: By definition, we know that a TD(1) language is a regular tree language. We need to show that there exists a regular tree language that is not a TD(1) language. For this we show that there exists a non-redundant regular tree grammar G, which is not a TD(1) grammar. This shows that $L(G)$ is a regular tree language, which is not a TD(1) language. Consider a non-redundant regular tree grammar with a content model, whose corresponding marked model group is (a_1^*, b_1^*, a_2^*). ♦

Theorem 8: *Single-type constraint languages form a proper subclass of* TD(1) *languages.* ■

Proof: A single-type constraint language is a TD(1) language by definition. Now consider a non-redundant regular tree grammar with a content model, whose corresponding marked model group is (a_1^*, b_1, a_2^*). ♦

Theorem 9: TDLL(1) *languages form a proper subclass of* TD(1) *languages.* ■

Proof: A TDLL(1) language is a TD(1) language by definition. Now consider a non-redundant regular tree grammar with a content model, whose corresponding marked model group is (a_1^*, a_2) ♦

Theorem 10: *Local tree languages form a proper subclass of single-type constraint languages.* ■

Proof: From definition, a local tree language is also a single-type constraint language. Proving the proper inclusion is straight forward, consider any non-redundant regular tree grammar with two symbols $A, B \in N1$, such that $A \neq B$, and *root*$(A) =$ *root*(B). Such a grammar is given in Example 8. ♦

Theorem 11: TDLL(1) *languages are not a proper subclass of single-type constraint languages.* ■

Proof: Consider the non-redundant regular tree grammar with a content model, whose corresponding marked model group is (a_1^*, b_1, a_2^*). ♦

Theorem 12: *Single-type constraint languages are not a proper subclass of* TDLL(1) *languages.* ■

Proof: Consider the non-redundant regular tree grammar with a content model, whose corresponding marked model group is (a_1^*, b_1^*, a_1^*).[8] ⧫

5 Boolean Set Operations on Tree Languages

In this section, we study the closure properties of the different language classes under boolean set operations: union, intersection and difference. Regular tree languages are closed under all the boolean set operations [CDG$^+$97]. We show that all the other language subclasses - TD(1), single-type constraint, TDLL(1) and local tree languages are closed under intersection, and not closed under union and difference.

To show closure under intersection, we introduce automata for regular tree languages. Because of the vertical and horizontal navigation for trees, we have to study two different automata - finite tree automata [CDG$^+$97] for the vertical navigation, and Glushkov automata [BKW98, BEGO71] for the horizontal navigation and ambiguities in model groups. We first give an introduction to these automata, define the automaton for the intersection language, and then proceed to study the closure properties.

5.1 Finite Tree Automata

For our requirements, we will use non-deterministic top-down tree automata, which we define below. We will define the automaton for a language with a normalized regular tree grammar representation $G = (N1, N2, T, P1, P2, S)$.(We will revisist tree automata in Section 7.)

Definition 20. (Non-deterministic Top-Down Finite Tree Automaton) [CDG$^+$97] A non-determinisitic top-down finite tree automaton T_A is defined by a 4-tuple, $T_A = (Q, \Sigma, q_I, \delta)$, where

- Q is the set of states. There is a state corresponding to every symbol in $N1$. We denote each state by the corresponding symbol in $N1$. Therefore $Q = N1$. (Note that $root(q)$ is defined for every state $q \in Q$.)
- $\Sigma = T$
- $q_I \subseteq Q$ is the set of initial states, there is a state in q_I for each symbol in S. Therefore $q_I = S$.
- $\delta : \Sigma \times Q \to MG(Q)$ is the transition function. In other words, the transition function gives a model group over Q given a symbol in Σ and a state in Q. We have a transition function corresponding to each production rule in $P1$. For example, for the rule $B \to b\ (A1^*, A2^*)$, the corresponding transition will be $\delta(b, B) = (A1^*, A2^*)$. □

Note that the top-down finite tree automaton corresponds directly to a regular tree grammar. Also the automaton does not define final states. The above automaton is non-deterministic. For example, consider the production rule $B \to b(A1^*, A2^*)$, where $root(A1) = root(A2) = a$. Let $b\langle a\rangle$ be a tree fragment that satisfies the above production rule. Now $\delta(b\langle a\rangle, B)$ can be either $b\langle A1(a)\rangle$ or $b\langle A2(a)\rangle$.

The tree-locality constraint for regular tree grammars is translated to tree automaton as follows: *for any symbol $t \in \Sigma$, there is at most one state $q \in Q$, such that $root(q) = t$.* Regular tree

[8] TDLL(1) languages without the deterministic content model constraint are actually a superclass of single-type constraint languages.

languages are closed under intersection [CDG+97]. Below we give the construction of the top-down tree automaton representing the intersection of two regular tree languages that are accepted by the two automata A_1 and A_2 respectively.

Definition 21. (Intersection Automaton for Regular Tree Languages) Let $A_1 = (Q_1, \Sigma_1, q_{I_1}, \delta_1)$ and $A_2 = (Q_2, \Sigma_2, q_{I_2}, \delta_2)$ be the two top-down non-deterministic finite tree automata representing the two languages. Let $A_3 = (Q_3, \Sigma_3, q_{I_3}, \delta_3)$ be the automaton that accepts the intersection language, in other words $L(A_3) = L(A_1) \cap L(A_2)$. We define A_3 as follows.

- Q_3 is defined as $\langle q_1, q_2 \rangle$, for all $q_1 \in Q_1, q_2 \in Q_2$, and $root(q_1) = root(q_2)$. We define $root(\langle q_1, q_2 \rangle) = root(q_1) = root(q_2)$.
- $\Sigma_3 = \Sigma_1 \cap \Sigma_2$.
- q_{I_3} is the set of states $\langle q_1, q_2 \rangle$, for all $q_1 \in q_{I_1}$, and $q_2 \in q_{I_2}$, and $root(q_1) = root(q_2)$.
- $\delta_3(t, \langle q_1, q_2 \rangle)$ is defined for all $q_1 \in Q_1, q_2 \in Q_2$, where $root(q_1) = root(q_2) = t \in \Sigma_3$ as follows: Let $\delta_1(t, q_1) = E_1$, and $\delta_2(t, q_2) = E_2$. Now $\delta_3(\langle q_1, q_2 \rangle) = E_3$, where $E_3 = E_1 \cap E_2$. Note that E_1 and E_2 are marked model groups. In the next subsection, we show how to obtain the marked regular expression E_3 whose alphabet consists of all symbols of the form $\langle a_1, a_2 \rangle$, where $a_1 \in Alphabet(E_1), a_2 \in Alphabet(E_2)$, and $h(a_1) = h(a_2)$, Therefore E_3 will be a model group over Q_3. □

One can obtain a regular tree grammar in NF1 from the intersection automaton. This is done by just reversing the steps in Definition 20, and then separating out the $P1$ and $P2$ rules.

5.2 Glushkov Automata for Marked Model Groups

In this subsection, we define Glushkov automata for general marked model groups. From [BKW98, BEGO71], we have one state in a Glushkov automaton that corresponds to one letter in the model group and we have a start state. The input alphabet of the automaton is the alphabet in the corresponding fully marked model group. We call this as a Glushkov automaton corresponding to a fully marked model group. For a general marked model group, we define a Glushkov automaton as follows (The only difference between the two Glushkov automata is in the input alphabet):

Definition 22. (Glushkov Automaton for a Marked Model Group) [BKW98] Consider a marked model group, E', whose corresponding fully marked model group is E'', and unmarked model group is E. In other words, $fm(E') = fm(E) = E''$, $h(E') = h(E'') = E$, and $h'(E'') = E'$. The Glushkov automaton for E' is defined as $A' = (Q_{E'}, \Sigma', \delta_{E'}, q_I, F_{E'})$, where

- $Q_{E'}$ is the set of states. There is one state corresponding to a letter in E'. We denote each state by the corresponding symbol in E''. In other words, $Q_{E'} = Alphabet(E'') \cup q_I$. Note that $h' : Q_{E'} \to Alphabet(E')$, and $h : Q_{E'} \to Alphabet(E)$ are defined for every state in $Q_{E'}$.
- $\Sigma' = Alphabet(E')$. Note that $h : \Sigma' \to Alphabet(E)$ is defined for every symbol in Σ'.
- q_I is the start state.
- $F_{E'}$ is the set of final states, obtained as defined in [BKW98].
- $\delta_{E'} : Q_{E'} * \Sigma' \to 2^{Q_{E'}}$ is the transition function. For $q \in Q_{E'}, a_i \in \Sigma'$, we define $\delta_{E'}(q, a_i) = \{y \mid y \in follow(E'', a_i)$, and $h'(y) = a_i\}$. □

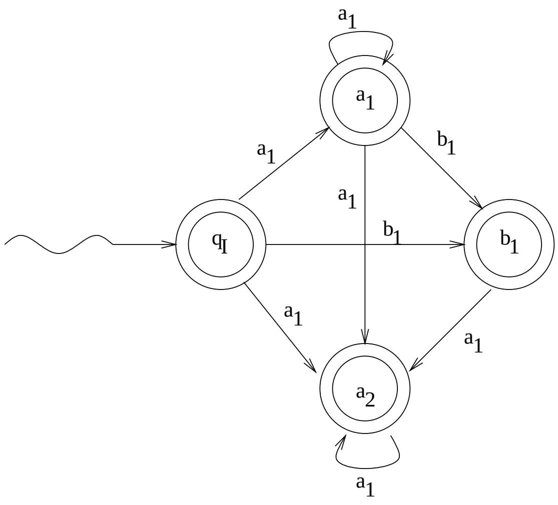


Figure 2: Glushkov automaton for the marked model group $E' = (a_1^*, b_1?, a_1^*)$, where $E'' = (a_1^*, b_1?, a_2^*)$, and $E = (a^*, b?, a^*)$

An example Glushkov automaton for the marked model group $E' = (a_1^*, b_1?, a_2^*)$ is shown in Figure 2. It is easy to observe the following for a Glushkov automaton.

- Just like the top-down finite tree automaton corresponds directly to a regular tree grammar in NF1, the Glushkov automaton corresponds directly to a model group.
- Consider a path (with possibly repeating edges) of states from q_I to a final state. This gives a string $s'' \in L(E'')$. For example, for the Glushkov automaton in Figure 2, there is a path of states from q_I, (a_1, b_1, a_2), where $(a_1, b_1, a_2) \in L(E'')$.
- A string s is accepted by A' if and only if $s \in L(E')$. Now s corresponds to a path of input symbols from q_I to a final state. A path of input symbols in the Glushkov automaton in Figure 2 is (a_1, b_1, a_1) and $(a_1, b_1, a_1) \in L(E')$.

Note that the unmarking function h is defined for the path of input symbols as well as the path of states. The function h' is defined for the path of states. For a path of states, say p, $h'(p)$ gives the corresponding path of input symbols. For example, $p = (a_1, b_1, a_2)$ is a path of states and $h'(p) = (a_1, b_1, a_1)$ is the corresponding path of input symbols. We define ambiguities in A' using $L(A')$, the language accepted by A'. Now $L(A') = L(E')$. Therefore ambiguities in E' are preserved in A'.

Definition 23. (Intersection Automaton for Marked Model Groups) Let $A_1 = (Q_{E_1'}, \Sigma_{1'}, \delta_{E_1'}, q_{I_1}, F_{E_1'})$ and $A_2 = (Q_{E_2'}, \Sigma_{2'}, \delta_{E_2'}, q_{I_2}, F_{E_2'})$ be the Glushkov automaton representing the two model groups. We define the intersection Glushkov automaton, $A_3 = (Q_{E_3'}, \Sigma_{3'}, \delta_{E_3'}, q_{I_3}, F_{E_3'})$ as follows:

- $Q_{E_3'}$ is defined as $\langle q_1, q_2 \rangle$, for all $q_1 \in Q_{E_1'}, q_2 \in Q_{E_2'}$, and $h(q_1) = h(q_2)$. We define $h'(\langle q_1, q_2 \rangle) = \langle h'(q_1), h'(q_2) \rangle$, and $h(\langle q_1, q_2 \rangle) = \langle h(q_1), h(q_2) \rangle$.
- $\Sigma_{3'} = \langle a_1, a_2 \rangle$, for all $a_1 \in \Sigma_{1'}, a_2 \in \Sigma_{2'}$, and $h(a_1) = h(a_2)$. We define $h(\langle a_1, a_2 \rangle) = h(a_1) = h(a_2)$.
- $q_{I_3} = \langle q_{I_1}, q_{I_2} \rangle$.

- $F_{E'_3} = \langle q_1, q_2 \rangle$, for all $q_1 \in F_{E'_1}, q_2 \in F_{E'_2}$, and $h(q_1) = h(q_2)$.
- $\delta_{E'_3}(\langle q_1, q_2 \rangle, \langle a_1, a_2 \rangle)$, where $\langle q_1, q_2 \rangle \in Q_{E'_3}$, and $\langle a_1, a_2 \rangle \in \Sigma_{3'}$) is defined as follows: $\langle r_1, r_2 \rangle \in \delta_{E'_3}(\langle q_1, q_2 \rangle, \langle a_1, a_2 \rangle)$, where $\langle r_1, r_2 \rangle \in Q_{E'_3}$ if and only if $r_1 \in \delta_{E'_1}(q_1, a_1)$, and $r_2 \in \delta_{E'_2}(q_2, a_2)$.

□

Given the intersection Glushkov automaton, we can obtain the marked regular expression using the flow-graph algorithm [BEGO71].

We define the intersection model group $E_3 = E_1 \cap E_2$, where E_1, E_2 are model groups, as the marked regular expression corresponding to the Glushkov automaton A_3 which represents the intersection of the two Glushkov automaton $A_1 = (\mathit{Alphabet}(fm(E_1)) \cup q_{I_1}, \mathit{Alphabet}(E_1), q_{I_1}, F_{E_1}, \delta_{E_1})$ and $A_2 = (\mathit{Alphabet}(fm(E_2)) \cup q_{I_2}, \mathit{Alphabet}(E_2), q_{I_2}, F_{E_2}, \delta_{E_2})$, where $F_{E_1}, F_{E_2}, \delta_{E_1}, \delta_{E_2}$ are defined appropriately.

We use the above step for taking intersection of two model groups while taking intersection of two tree automata. With this understanding of tree automata and Glushkov automata, we are ready to examine the closure properties of the various language classes which we introduced in Section 4.

5.3 Closure Properties

As mentioned before, regular tree languages are closed under union, intersection and difference. We show that all the subclasses of regular tree languages are not closed under union and difference. For showing that these languages are not closed, we give an example of two languages that are local tree as well as TDLL(1), such that the union (or difference) language can be captured by a *non-reducible regular tree grammar*, which is not a TD(1) grammar. (Note that this is not a non-redundant regular tree grammar, but it is easy to verify that the resulting language has no TD(1) grammar representation.) Finally we show that each language subclass is closed under intersection.

Lemma 5: There exist two grammars, G_1, G_2 that are both a local tree grammar as well as a TDLL(1) grammar, such that $L(G_1) \cup L(G_2)$ can be derived from a non-reducible regular tree grammar that is not TD(1). ■

Proof: Let G_1 and G_2 be regular tree grammars Let G_1 have the production rules $\{Doc \rightarrow doc\ (Para^*), Para \rightarrow para\ (\epsilon)\}$. Let G_2 have the production rules $\{Doc \rightarrow doc\ (Para^*), Para \rightarrow para\ (Fig)\}$. The union $L(G_1) \cup L(G_2)$ can be represented by the regular tree grammar G_3 with production rules $\{Doc \rightarrow doc\ (Para_1^* + Para_2^*), Para_1 \rightarrow para\ (\epsilon), Para_2 \rightarrow para\ (Fig)\}$. G_3 is not a TD(1) grammar. It is easy to verify that there is no TD(1) grammar that derives $L(G_3)$. ⧫

Theorem 13: TD(1), *single-type constraint,* TDLL(1), *and local tree languages are not closed under union.* ■

Lemma 6: There exist two grammars, G_1, G_2 that are both a local tree grammar as well as a TDLL(1) grammar, such that $L(G_1) - L(G_2)$ can be derived by a non-reducible regular tree grammar that is not TD(1). ■

Proof: Let G_1 have the production rules $\{Doc \rightarrow doc\ (Para^*), Para \rightarrow para\ (Fig?)\}$. Let G_2 have the production rules $\{Doc \rightarrow doc\ (Para^*), Para \rightarrow para\ (\epsilon)\}$. The difference $L(G_1) - L(G_2)$ is the the set of documents with at least one para with figures. This can be represented by the regular tree grammar G_3 with production rules $\{Doc \rightarrow doc\ (Para_1^*, Para_2, Para_1^*), Para_1 \rightarrow$

$para\ (Fig?), Para_2 \rightarrow para\ (Fig)\}$. G_3 is not a $\mathsf{TD}(1)$ grammar. It is easy to verify that there is no $\mathsf{TD}(1)$ grammar that derives $L(G_3)$. ♦

Theorem 14: $\mathsf{TD}(1)$, *single-type constraint,* $\mathsf{TDLL}(1)$, *and local tree languages are not closed under difference.* ■

Now we are ready to show the closure under intersection of local tree, single-type constraint, $\mathsf{TD}(1)$ and $\mathsf{TDLL}(1)$ languages. We will assume the following: L_1 and L_2 are the two regular tree languages, whose intersection is the regular tree language L_3. Let $A_1 = (Q_1, \Sigma_1, q_{I_1}, \delta_1)$ and $A_2 = (Q_2, \Sigma_2, q_{I_2}, \delta_2)$ be the top-down non-deterministic tree automata obtained from the normalized regular tree grammar representation of L_1 and L_2 respectively, that is, $L(A_1) = L_1$, and $L(A_2) = L_2$. Let $A_3 = (Q_3, \Sigma_3, q_{I_3}, \delta_3)$ represent the intersection tree automaton obtained from the construction as in Definition 21, such that $L(A_3) = L_3$.

Theorem 15: *Local tree languages are closed under intersection.* ■

Proof: L_1 and L_2 are local tree languages. Therefore, from definitions of the intersection tree automaton, and local tree languages, there is at most one state $q_3 = \langle q_1, q_2 \rangle \in Q_3$ such that $root(q_3) = root(q_1) = root(q_2)$. Therefore L_3 is a local tree language. ♦

For the remaining closure proofs, we have to consider the intersection of two marked model groups. We will assume the following: Let E_1 and E_2 be the two marked model groups, and we want to find the marked model group E_3 representing the intersection. Let $B_1 = (Q_1, \Sigma_1, \delta_1, q_{I_1}, F_1)$ and $B_2 = (Q_2, \Sigma_2, \delta_2, q_{I_2}, F_2)$ be the Glushkov automata for E_1 and E_2 respectively. Let the intersection Glushkov automaton be $B_3 = (Q_3, \Sigma_3, \delta_3, q_{I_3}, F_3)$, such that E_3 can be obtained from B_3 using the flow-graph algorithm.

Theorem 16: *Single-type constraint languages are closed under intersection.* ■

Proof: We have to show that when E_1 and E_2 satisfy the single-type type constraint, E_3 also satisfies the single-type constraint. We know from definitions of single-type constraint languages and intersection Glushkov automaton that there is at most one symbol $a_3 = \langle a_1, a_2 \rangle \in \Sigma_3$, such that $h(a_3) = h(a_1) = h(a_2)$. ♦

Theorem 17: $\mathsf{TD}(1)$ *languages are closed under intersection.* ■

Proof: We show that if E_1 and E_2 are unambiguous, then E_3 is unambiguous. Assume the contradiction, that is, let E_3 be ambiguous. We show that either E_1 or E_2 is ambiguous.

As E_3 is ambiguous, there exist two paths of input symbols, $p_1, p_2 \in L(E_3)$, where $p_1 = (\langle a_{1_1}, a_{2_1} \rangle, \langle a_{1_2}, a_{2_2} \rangle, \ldots, \langle a_{1_n}, a_{2_n} \rangle)$ and $p_2 = (\langle b_{1_1}, b_{2_1} \rangle, \langle b_{1_2}, b_{2_2} \rangle, \ldots, \langle b_{1_n}, b_{2_n} \rangle)$, and there exist two symbols $\langle a_{1_i}, a_{2_i} \rangle$ and $\langle b_{1_i}, b_{2_i} \rangle$, such that $\langle a_{1_i}, a_{2_i} \rangle \neq \langle b_{1_i}, b_{2_i} \rangle$, $h(\langle a_{1_i}, a_{2_i} \rangle) = h(\langle b_{1_i}, b_{2_i} \rangle)$, $h(\langle a_{1_1}, a_{2_1} \rangle, \langle a_{1_2}, a_{2_2} \rangle, \ldots, \langle a_{1_{i-1}}, a_{2_{i-1}} \rangle) = h(\langle b_{1_1}, b_{2_1} \rangle, \langle b_{1_2}, b_{2_2} \rangle, \ldots, \langle b_{1_{i-1}}, b_{2_{i-1}} \rangle)$, and $h(\langle a_{1_{i+1}}, a_{2_{i+1}} \rangle, \langle a_{1_{i+2}}, a_{2_{i+2}} \rangle, \ldots, \langle a_{1_n}, a_{2_n} \rangle) = h(\langle b_{1_{i+1}}, b_{2_{i+1}} \rangle, \langle b_{1_{i+2}}, b_{2_{i+2}} \rangle, \ldots, \langle b_{1_n}, b_{2_n} \rangle)$.

From definition of intersection automaton, we know that $p_{1_1} = (a_{1_1}, a_{1_2}, \ldots, a_{1_n})$, $p_{2_1} = (b_{1_1}, b_{1_2}, \ldots, b_{1_n}) \in L(E_1)$, and $p_{1_2} = (a_{2_1}, a_{2_2}, \ldots, a_{2_n})$, $p_{2_2} = (b_{2_1}, b_{2_2}, \ldots, b_{2_n}) \in L(E_2)$. From definition of Σ_3, this implies that either E_1 or E_2 is ambiguous. ♦

Theorem 18: $\mathsf{TDLL}(1)$ *languages are closed under intersection.* ■

Proof: We show that if E_1 and E_2 are 1-unambiguous, then E_3 is 1-unambiguous. Assume the contradiction, that is, let E_3 be 1-ambiguous. We show that either E_1 or E_2 is 1-ambiguous.

As E_3 is 1-ambiguous, there exist two paths of input symbols, $p_1, p_2 \in L(E_3)$, where $p_1 = (\langle a_{1_1}, a_{2_1}\rangle, \langle a_{1_2}, a_{2_2}\rangle, \ldots, \langle a_{1_n}, a_{2_n}\rangle)$ and $p_2 = (\langle b_{1_1}, b_{2_1}\rangle, \langle b_{1_2}, b_{2_2}\rangle, \ldots, \langle b_{1_p}, b_{2_p}\rangle)$, and there exist two symbols $\langle a_{1_i}, a_{2_i}\rangle$ and $\langle b_{1_i}, b_{2_i}\rangle$, such that $\langle a_{1_i}, a_{2_i}\rangle \neq \langle b_{1_i}, b_{2_i}\rangle$, $h(\langle a_{1_i}, a_{2_i}\rangle) = h(\langle b_{1_i}, b_{2_i}\rangle)$, $h(\langle a_{1_1}, a_{2_1}\rangle, \langle a_{1_2}, a_{2_2}\rangle, \ldots, \langle a_{1_{i-1}}, a_{2_{i-1}}\rangle) = h(\langle b_{1_1}, b_{2_1}\rangle, \langle b_{1_2}, b_{2_2}\rangle, \ldots, \langle b_{1_{i-1}}, b_{2_{i-1}}\rangle)$.

From definition of intersection automaton, we know that $p_{1_1} = (a_{1_1}, a_{1_2}, \ldots, a_{1_n})$, $p_{2_1} = (b_{1_1}, b_{1_2}, \ldots, b_{1_p}) \in L(E_1)$, and $p_{1_2} = (a_{2_1}, a_{2_2}, \ldots, a_{2_n})$, $p_{2_2} = (b_{2_1}, b_{2_2}, \ldots, b_{2_p}) \in L(E_2)$. From definitions of Σ_3, this implies that either E_1 or E_2 is 1-ambiguous. ♦

6 Evaluating Different XML Schema Language Proposals

In this section, we compare five representative XML schema language proposals: DTD, DSD, XML-Schema, XDuce, and RELAX. All these five schema proposals define a regular tree grammar. Converting the rules in these schema proposals to a regular tree grammar as in Definition 1 is not straighforward. Therefore we first give an equivalent definition of a regular tree grammar, and then explain the features in the schema proposals with respect to this regular tree grammar definition.

Definition 24. (Regular Tree Grammar) A regular tree grammar G is defined by a 6-tuple, $G = (N1, N2, T, P1, P2, S)$, where $N1, N2$ are non-terminal symbols, T is the set of terminal symbols, and $S \subseteq N1$ is the set of start symbols. $P1$ defines a set of production rules of the form $A \rightarrow aX$, where $A \in N1$, $a \in T$ and $X \in N2$. $P2$ defines a set of production rules of the form $X \rightarrow Expression$ where $Expression$ is a regular expression (model group) over symbols in $N1 \cup N2$. There can be multiple $P1$ or $P2$ rules for a symbol in $N1$ or $N2$, respectively. However, there is a restriction on $P2$ rules that the language defined by $contentModel(A)$, where A is any symbol in $N1$ is a regular language. □

A grammar as defined in the above section can be expressed as a grammar in Definition 1. The conversion requires that we identify a model group representing the content model of a non-terminal symbol. Such a model group exists because the content model of any symbol is a regular language.

6.1 DTD

DTD as defined in [BPE98] is a TDLL(1) and a local tree grammar. The tree-locality constraint is enforced by not distinguishing between terminal symbols and non-terminal symbols. The horizontal look ahead constraint is enforced during validating the DTD by constructing `followsets`, as mentioned in Appendix E of [BPE98].

6.2 DSD

Since DSD [KMS00] does not impose any constraint on the production rules, we can express any regular tree grammar in DSD. For example, $E = (a_1^*, p_1^*, a_2^*)$ is a valid content model in DSD. But the parsing in DSD uses a greedy technique 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 (a_1^m, a_2^n) which conforms to E.[9]

Element definitions in DSD have the form

```
<ElementDef ID="book-title" Name="title">
    SomeContentSpecification
</ElementDef>
```

[9] We conjecture that DSD accepts all and only TDLL(1) languages.

This can be converted into the grammar notation as

$$\begin{aligned} P1 &= \{book\_title \rightarrow title\ Book\_title\} \\ P2 &= \{Book\_title \rightarrow Expression\} \end{aligned}$$

Here, *Expression* is equivalent to `SomeContentSpecification`. Content patterns in DSD allow specifying content of an element, say A, based on even attribute values of A as well as those of its ancestors.

6.3 XML-Schema

XML-Schema represents TDLL(1) grammars that satisfy the single-type constraint. We define the different concepts in XML-Schema as follows.

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 grammar rule $Book \rightarrow (title, author^{+}, publisher?)$, where $Book \in N2$ and $\{title, author, publisher\} \subseteq N1$.

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.

   $$\begin{aligned} P1 &= \{ShipTo \rightarrow shipTo\ \ USAddress, BillTo \rightarrow billTo\ \ USAddress\} \\ P2 &= \{shipAndBill \rightarrow (ShipTo, BillTo)\} \end{aligned}$$

   XML-Schema allows a group definition to contain other group definitions without any restriction. Therefore, XML-Schema allows a context free language for the content model for an element as shown below:

   $$\begin{aligned} X &\rightarrow (A, Y, B) + \epsilon \\ Y &\rightarrow X \end{aligned}$$

   We believe that making content model description a context-free language was never the intention, but it was overlooked in [TBME00].[10]

[10] It is worth mentioning that the candidate recommendation for XML-Schema is believed to restrict the language defined by a content model to a regular language.

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

   $$P2 \quad = \quad \{Address \rightarrow (name, street, city), UKAddress \rightarrow (name, street, city, postcode)\}$$

   Now suppose there was an element declaration such as

   ```
   <element name="shipTo" type="Address"/>
   ```

   This is equivalent to the following two rules in P1 as shown below.

   $$P1 \quad = \quad \{ShipTo \rightarrow shipTo\ Address, ShipTo \rightarrow shipTo\ UKAddress\}$$

   Note that the rule $ShipTo \rightarrow shipTo\ UKAddress$ 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 does not allow to satisfy the horizontal and vertical lookahead constraints of a TDLL(1) grammar. For example, it is legal to have the following rules in XML-Schema.

   $$\begin{aligned} P2 \quad &= \quad \{X \rightarrow (Title), Y \rightarrow (Title, Author1^{+}), Z \rightarrow (Title, Author2^{+}, Publisher) \\ & \qquad AUTHOR1 \rightarrow Son^{*}, AUTHOR2 \rightarrow Daughter^{*}\} \\ P1 \quad &= \quad \{Title \rightarrow title\ TITLE, Author1 \rightarrow author\ AUTHOR1, \\ & \qquad Author2 \rightarrow author\ AUTHOR2, Book \rightarrow book\ X, Book \rightarrow book\ Y, Book \rightarrow book\ Z\} \end{aligned}$$

The above grammar does not satisfy the vertical and the horizontal lookahead constraints. But for checking document validity, the lookahead properties and the unique assignment tree 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 there is no way we can define a type such as $BOOK \rightarrow (Title, Author1^*, Author2^*)$ with $Author1 \rightarrow author\ A1$, $Author2 \rightarrow author\ A2$ in XML-Schema.

5. A substitution group definition (previously known as equivalence class) 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:

$$\begin{aligned} P1 &= \{ShipComment \rightarrow shipComment\ Y, customerComment \rightarrow cutomerComment\ Z \\ &\quad Ipo : comment \rightarrow shipComment\ Y, Ipo : comment \rightarrow shipComment\ Z\} \end{aligned}$$

   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 $Ipo : comment \rightarrow ipo : comment\ X$, and that Y, Z be derived from X.

6.4 XDuce

XDuce provides type definitions equivalent to a regular tree grammar. A type definition that produces a tree is converted into a rule in $P1$. Consider the example from [HVP00]: $type\ Addrbook = addrbook[Person^*]$ is written as $Addrbook \rightarrow addrbook(Person^*)$. Any type definition that does not produce a tree is written as $P2$ rules. For example, $type\ X = T, X \mid ()$ represents the $P2$ rules $X \rightarrow (T, X + \epsilon)$. 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*. This is slightly difficult to convert into grammar, because it is based on terminal symbols in T, and not on non-terminal symbols in $N1$. Below, we explain how we can convert the subtagging declarations into grammar rules. For example, consider the subtagging declaration

```
subtag i <: fontstyle
```

Consider all $P1$ rules that produce $fontstyle$ as the root of the production rule. Let these rules be $\{A \rightarrow fontstyle\ A1, B \rightarrow fontstyle\ B1, \ldots, N \rightarrow fontstyle\ N1\}$. Now introducing the subtag declaration adds the following additional rules to $P1$: $\{A \rightarrow i\ A1, B \rightarrow i\ B1, \ldots, N \rightarrow i\ N1\}$

6.5 RELAX

Any regular tree grammar can be expressed in RELAX [Mur00b, ISO00]. Let us first consider only `elementRule` and `hedgeRule` in RELAX. In other words we assume no tag names. This makes the role attribute in RELAX `elementRule`s equal to the tag name.

1. An `elementRule` defines a rule in $P1$ and a rule in $P2$. For example consider the `elementRule`, slightly modified from the one in [Mur00b].

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

   This can be converted into the production rules

   $$\begin{aligned} P1 &= \{Section \rightarrow section\ SECTION\} \\ P2 &= \{SECTION \rightarrow (paraWithFNotes^*)\} \end{aligned}$$

2. A `hedgeRule` defines a rule in $P2$. For example, consider the `hedgeRule` taken from [Mur00b]

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

   The above `hedgeRule` can be converted into grammar as

   $$P2 = \{blockElem \rightarrow (para)\}$$

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

3. RELAX allows 2 `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

   $$P2 = \{blockElem \rightarrow (para), blockElem \rightarrow (itemizedList)\}$$

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>
```

These three `elementRules` can be converted into three rules in $P1$ as

$$P1 \;=\; \{A \rightarrow a\ X, A \rightarrow a\ Y, A \rightarrow b\ Z\}$$

5. `Tag` is used to typically specify attributes. In effect it 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

$$P1 \;=\; \{Val \rightarrow val\ X, Val \rightarrow val\ Y\}$$

Figure 3 compares the expressive power of the different grammar classes and XML schema language proposals.

7 Membership Checking and Type Assignment

In this section, we study two operations performed given a grammar (as defined in Definition 2) and a tree:

- Membership checking (Document validity checking): determines whether the tree is permitted by the grammar.
- Type or non-terminal assignment (Document interpretation): determines an assignment tree for the tree.

It is necessary to distinguish between the tree model and event model. Both models work fine for small documents. When documents are significantly large, the tree model has performance problems while the event model scales well.

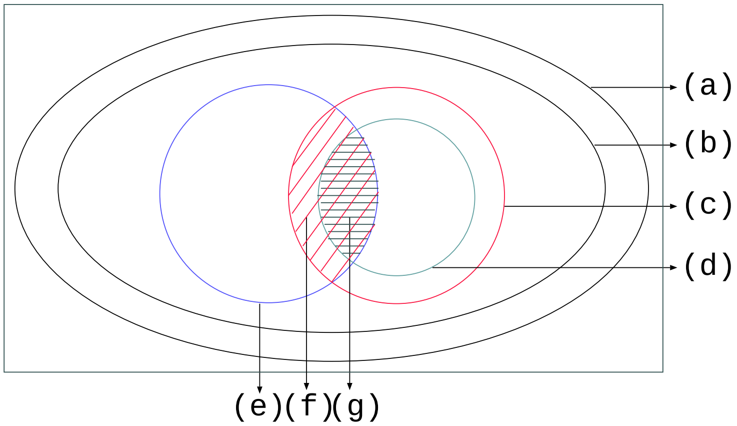


Figure 3: The expressive power of the different regular tree grammars: (a) regular tree grammars (RELAX, XDuce), (b) TD(1) grammars, (c) single-type constraint grammars, (d) local tree grammars, (e) TDLL(1) grammars, (f) TDLL(1) grammars with single-type constraint (XML-Schema), (g) TDLL(1) grammars with tree-locality constraint (DTD)

In the tree model, application programs have access to the tree in memory, and can traverse the tree any number of times. The tree may be created by the XML parser and accessed via some API such as DOM [WHA$^+$00]. Alternatively, the tree may be created by application programs from the output of the XML parser. In the event model, application programs are notified of the start or end element event via some API such as SAX [Meg00]. A start or end element event is raised when the XML parser encounters a start or end tag, respectively. In other words, application programs visit and leave elements in the document in the depth-first manner. Therefore once an application program visits an element in the document, it will not be able to visit the element again. We will assume that application programs will not attempt to reconstruct the document in memory or buffer events in memory.

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 accept only a subset of regular tree languages. 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. However, they become a lot more useful for XML, if they are allowed to examine an element before assigning a state to it. To refer to such automata, we use the phrase “with one lookahead”.

We do not consider complexity with respect to the size of XML schemas, but consider complexity with respect to the size of XML documents. The size of documents may be significantly large (e.g., 16 MB), but the size of schemas is rather limited. In fact, DTDs are considered large when they have more than 200 element types.

7.1 Membership Checking

First, we consider membership checking. We show that there is an efficient algorithm that is applicable to any regular tree grammar.

Theorem 19: *Membership checking for any regular tree grammar can be created in the event model, and the time required is linear to the number of nodes in the document.* ■

This algorithm simulates non-deterministic bottom-up tree automata. It assigns (zero, one, or possibly multiple) non-terminals in $N1$ to each element, after assigning non-terminals in $N1$ to each of the child elements (if any). Let e_0 be an element and $e_1, e_2, \dots, e_i$ be its child elements. A non-terminal, say A_0 ($\in N1$), is assigned to e_0 if there exists a $N1$ non-terminal sequence $A_1 A_2 \dots A_i$ such that

- for some production rule $A_0 \rightarrow a\ X_0$ in $P1$, a is the terminal symbol of e_0,
- for some production rule $X_0 \rightarrow \mathit{Expression}$ in $P2$, $A_1 A_2 \dots A_i$ matches $\mathit{Expression}$, and
- for every j $(1 \leq j \leq i)$, A_j is one of the non-terminals assigned to e_j.

We assume that a deterministic string automaton has been already constructed from the content model of each production rule. Given a non-terminal (in $N1$) of the regular tree grammar and a state of this deterministic string automaton, the transition function returns a new state. We show that this algorithm can be implemented in the event model. The key idea is to maintain a set of states for each string automaton. Sets of states are pushed into and popped from a stack when the XML parser raises a begin element or end element event, respectively. The idea is illustrated in Algorithm `Membership`. Obviously, the time complexity of this algorithm is linear to the number of nodes in the document.

Algorithm 1: `Membership`

begin element When a start tag is encountered, those production rules for the terminal symbol of this tag are identified. Note that more than one production rule may be found. Then, for each of these production rules, an empty set of states of the corresponding deterministic string automaton is created. The sets of states for the parent element are pushed into the stack.

end element When an end tag is encountered, each set of states is examined. If it contains a final state, the non-terminal of the corresponding production rule is assigned to this element. After assigning (zero, one, or multiple) non-terminals to this element, the sets of states for the parent element are popped from the stack. For each of these non-terminals and each current state in each set, the next state is computed by applying the transition function of the corresponding string automaton.

This algorithm can be applied to TD(1), TDLL(1), single-type constraint, and local tree grammars. Thus, membership checking is always efficient. However, as we see in the next subsection, there are other algorithms for such restricted grammars. These algorithms perform membership checking as well as type assignment, and they are arguably more straightforward.

7.2 Type Assignment

We first consider type assignment for local tree grammars, single-type constraint grammars, and TDLL(1) grammars. If and only if an assignment tree can be constructed for a given document and grammar, the document is permitted by the grammar. Thus, algorithms for type assignment can be used for membership checking.

Theorem 20: *Type assignment for a local tree grammar, single-type constraint grammar, or* TDLL(1) *grammar can be built in the event model, as a deterministic top-down tree automaton with one lookahead. The time required is linear to the number of nodes in the document.* ■

Our type assignment algorithms for these three classes of grammars are very similar. These algorithms simulate deterministic top-down tree automata with one lookahead. They assign a non-terminal to each element, after assigning a non-terminal to its parent element and elder sibling elements. Let e_0 be an element and $e_1, e_2, \ldots, e_i$ be its child elements. Suppose that the non-terminal assigned to e_0 is A_0 ($\in N1$), and that the non-terminals assigned to $e_1, e_2, \ldots, e_{j-1}$ are $A_1, A_2, \ldots, A_{j-1}$ ($\in N1$), respectively. Then, non-terminal A_j ($\in N1$) is assigned to e_j, if the following conditions hold:

- for some production rule $A_0 \rightarrow a\ X_0$ in $P1$, a is the terminal symbol of e_0,
- for some production rule $X_0 \rightarrow Expression$ in $P2$, $A_1 A_2 \ldots A_{j-1} \in L(prefixMG(Expression, A_j))$, and
- for some production rule $A_j \rightarrow b\ Y$ in $P1$, b is the terminal symbol of e_j.

We show that these algorithms can be implemented in the event model. That is, type assignment programs receive events and forward them to application programs; if a begin element event is received, the type of this element is added. We assume that a deterministic string automaton has been already constructed from the content model of each production rule. The key idea is to maintain a state of the deterministic string automata for the current element. The state for the current element is pushed into and popped from a stack when the XML parser raises a begin element or end element event, respectively. The idea is illustrated in Algorithm `TypeAssignment`. Obviously, the time complexity of this algorithm is linear to the number of nodes in the document.

Theorem 21: *The type assignment for a regular tree grammar may be ambiguous.* ■

Proof: A simple example for a regular tree grammar and a tree for which the type assignment is ambiguous is the following.

$$\begin{array}{rcl} Book & \rightarrow & book(Title, Author1^*, Author2^*) \\ Title & \rightarrow & title(Pcdata) \\ Author1 & \rightarrow & author(Son^*) \\ Author2 & \rightarrow & author(Daughter^*) \\ Pcdata & \rightarrow & pcdata(\epsilon) \end{array}$$

```
<book>
  <title><pcdata/></title>
  <author/>
</book>
```

Algorithm 2: `TypeAssignment`

begin element When a start tag is encountered, the non-terminal for this element is determined. Depending on the class of grammars, this is done as follows:

Case 1: local tree grammars A non-terminal is determined directly from the terminal symbol in the current start tag (see Definition 7).

Case 2: single-type constraint tree grammar From the non-terminal A ($\in N1$) assigned to the parent element, a content model, $\mathit{contentModel}(A)$, is found. This content model contains at most one non-terminal A' such that $\mathit{root}(A')$ is the terminal symbol of the current start tag (see Definition 13). If such a non-terminal is found, it is the non-terminal for the current element. Otherwise, type assignment fails.

Case 3: $\mathsf{TDLL(1)}$ grammars We assume that deadend states (i.e., states from which final states cannot be reached) have been removed from deterministic string automata for content models. Now, consider the current state of the string automaton for the parent element. Transitions from this state have associated non-terminals in $N1$. The $\mathsf{TDLL(1)}$ constraint (Definition 16) ensures that at most one of these non-terminals, say A, has the terminal symbol in the current start tag as $\mathit{root}(A)$. If such a non-terminal is found, it is the non-terminal for the current element. Otherwise, type assignment fails.

Then, the current state for the parent element is updated. The next state is computed by applying the transition function of the corresponding string automaton to the non-terminal of the current element and the current state for the parent element.

Next, the production rule for this element is identified. The initial state of the corresponding string automaton is then created as the state of the current element. The state for the parent element is pushed into the stack.

end element When an end tag is encountered, the state of the current element is examined. If it is not the final state of the corresponding string automaton, type assignment fails.

Now the node represented by ⟨`author/`⟩ can be assigned to either $Author1$ or $Author2$. Note that an ambiguous assignment is not possible for any $\mathsf{TD(1)}$, $\mathsf{TDLL(1)}$, single-type constraint, or local tree grammar. ♦

Theorem 22: *Type assignment for a* $\mathsf{TD(1)}$ *grammar cannot be built in the event model.* ■

Proof: Consider a grammar as follows:

$$\begin{aligned} Doc &\rightarrow doc(Para1?, (Para2, Para2)^*) \\ Para1 &\rightarrow para(Pcdata) \\ Para2 &\rightarrow para(Pcdata) \\ Pcdata &\rightarrow pcdata(\epsilon) \end{aligned}$$

The type of the first paragraph can be determined only after examining the last paragraph. If a document contains an odd number of paragraphs, the type of the first paragraph is $Para1$. If it contains an even number of paragraphs, the type is $Para2$. However, the event model cannot buffer the begin element event for the first paragraph. ♦

7.3 Implementations

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

- DTD: Membership checking against DTDs has been widely implemented. Most implementations are based on the event model, and they simulate top-down deterministic automata with one lookahead.
- XDuce: Type assignment implemented by the designers of XDuce is based on the tree model. This implementation simulates top-down non-deterministic automata by backtracking. Such backtracking may require exhaustive search.
- DSD: Type assignment of DSD is similar to that of XDuce. It is based on the tree model, and simulates top-down non-deterministic automata by backtracking. However, backtracking is not thorough, and thus does not perform exhaustive search.
- XML-Schema: Schema-validity assessment, as defined in 7.2 of XML-Schema Part 1, performs type assignment by using deterministic top-down tree automata with one lookahead. To the best of our knowledge, all implementations follow this model.

  Although XML-Schema allows TDLL(1) grammar satisfying the single-type constraint only, other mechanisms of XML-Schema (`key`, `unique`, and `keyref`) require use of the tree model.

  - XSV (XML-Schema Validator): This is based on the tree model.
  - 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`.
  - Xerces Java Parser: This implementation is based on the event model, and does not support `key`, `unique`, and `keyref`.
  - XML Spy and `XML Instance`: They are XML document editors, and thus use the tree model. Editing of documents can be controlled by schemas in XML-Schema.
- RELAX: Four implementations of RELAX are available [11]. They use different algorithms for type assignment and membership checking.
  - VBRELAX: This program is based on the tree model. It simulates top-down non-deterministic automata by backtracking. Such backtracking may require exhaustive search.
  - 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 `Membership`.
  - RELAX Verifier for Java: This program is based on the event model. It is also based on Algorithm `Membership`, but further utilizes top-down non-determinism. Intuitively

[11] They are available at the official site of RELAX, http://www.xml.gr.jp/relax.

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 TDLL(1) grammars without the deterministic constraint.

- Relaxer: This 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.

8 Conclusion

A mathematical framework using formal language theory to compare various XML schema languages is presented. In our framework, a normal form representation for regular tree grammars, ambiguities in general marked regular expressions and model groups, and various subclasses of regular tree languages are defined. Further, a detailed analysis of the closure properties and expressive power of the different subclasses is presented. Finally, results on the complexity of membership checking and type resolution for various XML schema languages are presented.

One class of grammars which we did not describe in great detail is TDLL(1) grammars without deterministic constraint. However, we believe that this class of grammars will play an important role because of its several useful features – (a) membership checking and type assignment can be done in linear time in the event model, and (b) it is strictly more expressive than TDLL(1) grammars with deterministic constraint and single-type constraint grammars, though strictly less expressive than TD(1) grammars.

We expect a future XML processing system will behave as follows: A server does XML processing, and the result of XML processing is an XML document and a regular tree grammar, as in XDuce. The server will try to evaluate if the regular tree grammar has an equivalent representation as, say, a TDLL(1) grammar. If yes, it will convert the grammar to that form. The server will then send the document and the grammar to the client. Now the client gets a document and a TDLL(1) or regular tree grammar. The client might wish to do more processing, but might be limited by memory considerations. If the grammar the client gets is a regular tree grammar, and the client has memory limitations, it will try to convert the grammar into the "tightest" possible TDLL(1) grammar, and then do the processing.

Though our work provides the framework for such systems, there are several problems that are still to be answered before we can come up with good solutions – for example, what is the "tightest", say, TDLL(1) grammar for a given regular tree grammar? There are several other interesting research topics, one of them is k lookaheads in the vertical and horizontal directions. Also the algorithmic aspects of type checking still pose lot of challenges – for example, what is the "precise" class of grammars for which we can do type assignment in the event model? Also error recovery and error messages are important to implementations.

References

[AU79] A. V. Aho and J. D. Ullman. *"Principles of Compiler Design"*. Addison–Wesley, 1979.

[BEGO71] R. Book, S. Even, S. Greibach, and G. Ott. "Ambiguity in Graphs and Expressions". *IEEE Transactions on Computers*, 20(2):149–153, Feb. 1971.

[BKW98] A. Bruggemann-Klein and D. Wood. "One-Unambiguous Regular Languages". *Information and Computation*, 140:229–253, 1998.

[BPE98] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds.). *"Extensible Markup Language (XML) 1.0"*. W3C, Feb. 1998. http://www.w3.org/TR/REC-xml.

[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. "XML Schema Part 0: Primer", Sep. 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.

[ISO00] ISO/IEC. *Information Technology – Text and Office Systems – Regular Language Description for XML (RELAX) – Part 1: RELAX Core*, 2000. DIS 22250-1.

[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.

[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)", 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.

[TBME00] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn (Eds.). “XML Schema Part 1: Structures”, Sep. 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/.