Ambiguity Detection Of RELAX Grammars

KAWAGUCHI Kohsuke, (kohsukekawaguchi@yahoo.com)
Sun Microsystems, Inc.

May 24, 2001

Abstract

In this paper, an ambiguity detection algorithm for grammars written in RELAX, a schema language for XML, is presented. Ambiguous schemata cause troubles to those applications which process document instances written in them. This algorithm is a sound as well as complete algorithm to determine the ambiguity of a grammar.

1 Introduction

The proposed algorithm is an ambiguity detection algorithm for general hedge regular grammar, with application to RELAX [1] in mind. RELAX is a grammar-based schema language for XML.

Modern schema languages, like RELAX and TREX [2], are powerful in terms of expressiveness, compared to older schema languages like DTD. For example, they allow non-deterministic content models, tag name overloading, more powerful datatyping, and so on [5]. However, at the same time, this expressive power causes some problems. One of these problems is the ambiguity. Consider the following RELAX grammar:

```
<elementRule label="root">
  <tag />
  <ref label="foo" occurs="+" />
</elementRule>

<elementRule label="foo">
  <tag>
    <attribute name="id" type="ID" />
    <element name="ref" type="IDREF" />
  </tag>
</elementRule>

<elementRule label="foo">
  <tag>
    <attribute name="id" type="string" />
    <element name="ref" type="string" />
  </tag>
</elementRule>
```

and let us see what happens when the following XML instance is validated by the above grammar.

```
<root>
  <foo id="a" ref="b" />
  <foo id="a" ref="a" />
  <foo id="b" ref="a" />
</root>
```

If the first `foo` is considered as ID/IDREF pair, then the second `foo` must be string/string pair, and the third must be ID/IDREF pair. Another possible interpretation is that all `foo` are string/string pair. And other interpretations are also possible.

Intuitively, to enforce ID/IDREF constraints in such a grammar is considerably harder than doing the same thing for DTD. Several schema languages, including XML Schema [3], avoids this problem by sticking to deterministic content model, but it severely limits the expressive power of the language.

Another problem is that ambiguous grammars make it difficult to further utilize the result of validation. For example, imagine a program that works on the interpretation of the above grammar. If more than one interpretation is possible, the program is confused. It has to employ some technique to resolve the ambiguity. In fact, every application program that accepts the result of validation will suffer the same problem.

The goal of this paper is to present the algorithm to detect whether the given grammar causes ambiguous interpretation for some instances that conform to it. The algorithm is sound as well as complete, which means it always computes the correct answer.

1.1 Outline of this paper

First, some concepts and definitions are provided, which is necessary to discuss the algorithm. In particular, ambiguity of grammar is formally defined. Next, ambiguity of labels and the concept of "catalyst" are introduced as a primitive building block of ambiguous grammar. Then, the algorithm itself is described in three separate sections. Various properties of the ambiguity are discussed. Proof of soundness and completeness is provided in appendices.

2 Generalized tree regular grammar

First, I'd like to introduce the notation of generalized tree regular grammar. The following is a very simple example.

$$\begin{aligned} \mathrm{D} \rightarrow &\quad \textbf{doc} < \mathrm{P}+ > \\ \mathrm{P} \rightarrow &\quad \textbf{para} < \textit{string} > \end{aligned}$$

This grammar is equivalent to the following RELAX grammar and TREX pattern.

- RELAX

```
<elementRule label="D" role="doc">
  <ref label="P" occurs="+" />
</elementRule>
<elementRule label="P" role="para" type="string" />
```

- TREX

```
<define name="D">
  <element>
    <ref label="doc" />
    <oneOrMore><ref label="P" /></oneOrMore>
  </element>
</define>
<define name="P">
  <element>
    <ref label="para" />
    <anyString />
  </element>
</define>
```

Grammar consists of *rules*. A rule (e.g.,$D \rightarrow$ **doc** $< P+ >$) consists of three parts; a *label* (e.g.,D), a *clause* (e.g.,**doc**), and a content model (e.g.,$P+$).

For now, clauses can be considered as a tag name of the element. Later, we will discuss how to extend clause to cover attributes. In this paper, clauses are denoted by bold words. Content models are string regular expression over labels and datatypes. In this paper, labels are denoted by upper case alphabets, and datatypes are denoted by italic words.

It is a proven fact that any RELAX grammar can be written in this notation [4]. And probably any TREX pattern can be written in this notation, too. And any grammar written in this notation can be expressed by both languages.

3 Ambiguous grammars and interpretations

Let us think about what an ambiguous grammar really is.

Say there is a grammar G, and an XML instance X which conforms to G. Intuitively, we can map definitions in G to all elements and attributes of X.

If G is a "well-written" grammar, this mapping can be done uniquely. That is, no two definitions are applicable to the same element while satisfying constraints imposed by G. However, if G has some "flaws" sometimes it is possible to map definitions to elements in more than one way. These grammars are what we consider ambiguous.

For example, consider the following grammar as G:

$$\begin{aligned} \mathrm{S} &\rightarrow \quad \textbf{start} < F_1 \mid F_2 > \\ F_1 &\rightarrow \quad \textbf{foo} < \mathrm{B} > \\ F_2 &\rightarrow \quad \textbf{foo} < \mathrm{B}* > \\ \mathrm{B} &\rightarrow \quad \textbf{bar} < \textit{string} > \end{aligned}$$

and the following as an instance X.

```
<start>
  <foo>
    <bar />
  </foo>
</start>
```

As you can see, both F_1 and F_2 is applicable to `foo` element. Therefore this grammar is ambiguous. Of course, there are some instances that have unique definition-element mapping.

```
<start>
  <foo>
    <bar /><bar />
  </foo>
</start>
```

3.1 Interpretation

From now on, I'd like to call this mapping from definitions to elements "interpretation". For example, "interpretation of X by G" means the label assignment to all elements of X, as shown in the following.

```
<start> : label=S, clause=start
  <foo> : label=F2, clause=foo
    <bar /> : label=B, clause=bar
    <bar /> : label=B, clause=bar
  </foo>
</start>
```

Simularly, "interpretation of element E by G" means the label/clause assignment to an element E and its descendants.

Furthermore, we can consider interpretation by a rule. "interpretation of element E by a rule $L \rightarrow \mathbf{R} < \ldots >$" means the label/clause assignment to an element E and its descendants, in which label L and clause $\mathbf{R}$ are assigned to E.

Definition 3.1 (Ambiguous grammar) Grammar G is said to be ambigous when there exists a valid instance X and at least two different interpretations of X by G.

3.2 Ambiguous labels

In this section, the ambiguous relation of two labels will be discussed and defined. This relation is important because this is an unit of grammar ambiguity. This relation is a relation between two *different* labels, and will be denoted as $A_1 \sim A_2$ in this paper.

Intuitively, two labels A_1 and A_2 are said to be in the ambiguous relation when there exists an XML tree that can be interpreted by A_1 as well as A_2. In aforementioned example, F_1 and F_2 are in the ambiguous relation, because `<foo><bar/></foo>` can be interpreted by both of them.

Definition 3.2 (Ambiguous relation of labels) Label A_1 and A_2 are in the ambiguous relation if and only if:

- there exists an XML tree that can be interpreted by A_1 as well as A_2, and
- these two interpretations are different.

3.3 Ambiguous labels and ambiguous grammars

Let us go back to think about an ambiguous grammar G again. Since G is an ambiguous grammar, we have X, which is an XML instance that can be interpreted by G in more than one way. We name one interpretation I, and another interpretation I'. By the definition of the ambiguous grammar, the existence of two such different interpretations (I and I') are trivial.

The fact that I and I' are different means that there exists at least one XML element E in X such that I and I' give different labels or clauses for this element. Therefore, I and I' have to give a different label for this element. Say interpretation I gives label A to E, and I' gives A' to E.

If we look at this situation from another point of view, we can say that A and A' are in ambiguous relation, because E can be interpreted by both A and A', and this is exactly the definition of label ambiguity.

Therefore, from the above reasoning, the following proposition is proved.

Proposition 3.3 When the grammar is ambiguous, there always exists at least one pair of labels that are in the ambiguous relation.

3.4 Catalyst

However, a grammar is sometimes unambiguous even if it has some ambiguous pairs of labels. The following grammar is one of such examples.

$S \rightarrow \quad \mathbf{e} < F, B >$
$F \rightarrow \quad \mathbf{e} <>$
$B \rightarrow \quad \mathbf{e} <>$

In this example, an XML tree `<e />` can be interpreted differently (by F or by B). Therefore $F \sim B$. However, in terms of grammar, this grammar accepts one instance `<e><e/><e/></e>` only and the interpretation for this instance is unique. Therefore, the grammar is still unambiguous.

When we think in terms of labels, F and B are actually in the ambiguous relation. However, other constraints prevent assigning those two labels to the same element.

And that's exactly why this grammar is not ambiguous, although it has ambiguous labels.

If I modify the grammar so that we can have a chance to choose, the modified grammar becomes ambiguous. See the following modified version.

$S \rightarrow \quad \mathbf{e} < F|B >$
$F \rightarrow \quad \mathbf{e} <>$
$B \rightarrow \quad \mathbf{e} <>$

This grammar is ambiguous because `<e><e/></e>` cannot be interpreted uniquely.

Intuitively, we can say that for a grammar to be ambiguous, we need the third label, which acts as catalyst. This catalyst provides a choice between two labels that are in the ambiguous relation. This is the key factor that makes a grammar ambiguous. From now on, I call this third label as "catalyst".

Formal definition of catalyst will be presented later. But first, for the concept of catalyst to be useful, it must satisfy the following property.

Proposition 3.4 A grammar is ambiguous if and only if it has at least one catalyst.

4 Outline of Algorithm

The following procedure describes the outline of the algorithm.

Algorithm 1 detect whether the grammar is ambiguous

Input: G : grammar to test
Variable: S : score matrix
{ iteratively compute ambiguity between labels }
repeat
  $f \Leftarrow$ `false`
  for all blank entries (i, j) in S **do**
    $S_{ij} \Leftarrow \mathrm{dTLA}(i, j, S)$ { compute ambiguity between two labels }
    if S_{ij} is not blank **then**
      $f \Leftarrow$ `true`
    end if
  end for
until $f =$ `false` { repeat until they converge }

{ mark blank entries by 'U'("unambiguous") }
for all blank entries (i, j) in S **do**
  $S_{ij} \Leftarrow$ "unambigous"
end for

{ find catalyst }
for all label i in G **do**
  if $\mathrm{dC}(i, S)$ outputs "i is a catalyst" **then**
    output "G is ambiguous"
    halt
  end if
end for
output "G is unambiguous"

4.1 Score matrix

"score matrix" is a two-dimensional array. If we have five labels (A,B,C,D, and E), then the score matrix is as follows.

B	C	D	E	
				A
				B
				C
				D

Note that there are no A vs A or no B vs B. Ambiguous relationship is defined only between two different labels. Each field can have three different states.

1. If the box is left blank, it means we don't know if those two are in the ambiguous relation or not.
2. When the box is marked by 'A', it indicates that those two are in the ambiguous relation.
3. And we'll mark it by 'U' when we find that those two are NOT in the ambiguous relation (unambiguous).

4.2 Iterative label ambiguity computation

Consider the following example grammar.

$F \rightarrow$ **foo** $< A >$
$F' \rightarrow$ **foo** $< B >$

We have two labels and we'd like to compute if these two are in ambiguous relation. Since we have the same clause for both labels, the ambiguity of these two solely depends on the ambiguity between A and B.

If we know S_{AB} = 'A' (A and B are in the ambiguous relation), we can say that F and F' are also in ambiguous relation. If we know S_{AB} = 'U', then they are also NOT in ambiguous relation. However, if S_{AB} is still blank, then this computation has to be postponed until S_{AB} is marked.

So as we see, sometimes the ambiguous relation between labels depends on that between other labels. Therefore, this step has to be done iteratively until no new information is discovered.

The algorithm dTLA is explained later.

4.3 Unmarked entries after iteration

The first iterative phase trivially converges, but it may leave some fields unmarked. This case is discussed later in detail. The bottom line is, we can safely assume that these labels are not ambiguous in this case.

4.4 Finding a catalyst

For every label in the grammar, this phase checks whether that label can be a catalyst or not. This algorithm (dC) will be also explained later.

By the definition of catalyst, if one is found, the entire grammar is ambiguous. If no such label is found, then the entire grammar is NOT ambiguous.

5 Algorithm to check the ambiguity between two labels

This section proposes an algorithm that detects ambiguity of two labels. In this paper, this algorithm will be called "detectTwoLabelsAmbiguity" (dTLA).

dTLA receives two labels to be tested and "score matrix" as defined in the previous section. This matrix is only partially complete when dTLA is used. So some fields could be blank. Sometimes information of score matrix is necessary to determine the ambiguous relation of L_1 and L_2, and that's why dTLA receives it as a parameter.

dTLA always produces one of the following outputs, and it always stops. [1]

[1] Intuitively, this algorithm computes intersection of automaton L_1 and automaton L_2. This is a very classic problem of computer science. If the intersection is the empty set, we know that these two labels are not in ambiguous relation. If the intersection accepts some alphabet sequences, then we know that these two are in ambiguous relation.

The only trick used here is to treat ambiguous labels equal. However, these ambiguity relation cannot be considered as an equality relation because it lacks transitivity.

- L_1 and L_2 are in the ambiguous relation
- L_1 and L_2 are NOT in the ambiguous relation
- Currently, the relation is undecidable.

Let me first formulate the algorithm by using pseudo code. After that, somewhat human-friendly explanation is presented.

Algorithm 2 dTLA

Input: L_1, L_2 : two labels to be tested
S : "score matrix"
for all rule $R_i : L_1 \rightarrow \mathbf{c} < M >$ whose label is L_1 **do**
for all rule $R_j : L_2 \rightarrow \mathbf{c}' < M' >$ whose label is L_2 **do**
{if two clauses are different, R_i and R_j cannot be ambiguous.}
if $\mathbf{c} = \mathbf{c}'$ **then**
Let automaton of M be (Q, S, T, q_0, F)
Let automaton of M' be (Q', S, T', q_0', F')

$S^* \Leftarrow \{(s, s'), s \in S, s' \in S | s = s' \vee s \sim s'\}$
$T^*\big((q, q'), (s, s')\big) = \big(T(q, s), T'(q', s')\big)$
$M^* \Leftarrow (Q \times Q', S^*, T^*, (q_0, q_0'), F \times F')$ { create a new automaton }

if the language accepted by M^* is not empty **then**
output "L_1 and L_2 are in the ambiguous relation"
exit function
end if
$S^+ \Leftarrow \{(s, s'), s \in S, s' \in S | s = s' \vee \neg s \not\sim s'\}$
$M^+ \Leftarrow (Q \times Q', S^+, T^*, (q_0, q_0'), F \times F')$ { create another automaton}

if the language accepted by M^+ is not empty **then**
output "currently, the relation is undecidable"
exit function
end if
end if
end for
end for
{We've tested all rule pairs}
output "L_1 and L_2 are not in the ambiguous relation"

Several rules can share the same label. Therefore, dTLA tests all possible pairs by enumerating rules that shares the label. Clause is tested first: if the clauses differ (e.g., $F \rightarrow \mathbf{x} < \cdots >$ and $G \rightarrow \mathbf{y} < \cdots >$, then those two rules can never be ambiguous because we can always tell the difference by looking at its tag name.

Next, the content model automata are of interest in this algorithm. Note that two automata share the same alphabet set S, which is the set of all labels in this grammar.

This algorithm will derive a new automaton M^*, where S^* is a subset of $S \times S$.

Let us assume that we have two labels X and Y and they are marked as ambiguous in the score matrix. For the automaton M and M', X and Y are alphabets. And by the definition of the label ambiguity, there exists a very nice XML fragment such that it can be interpreted by both X and Y. If we read such a fragment, we can perform a transition by X for M, and by Y for M' (or vice versa). M^* is an automaton that simulates this behavior. Its alphabet is $S \times S$ because one goes to M and another goes to M'.

If the language accepted by M^* is not empty, then the algorithm says "L_1 and L_2 are in the ambiguous relation" and stops.

Otherwise, the algorithm considers another automaton M^+, where S^+ is S^* plus those pairs whose relation is not yet determined. Intuitively, all undetermined relationships are treated as ambiguous in M^+. If two labels are unambiguous even if all undetermined are ambigous, then we are sure that these two labels are unambiguous. So if the language accepted by M^+ is the empty set, then the algorithm says "L_1 and L_2 are not in the ambiguous relation" and stops.

Otherwise, it says "currently, the relation is undecidable" and stops.

5.1 How dTLA works

Consider the following transition sequence performed by M^*. Note that s_1, s_1', s_2, s_2' are all labels.

$$(q_0, q_0') \xrightarrow{(s_1,s_1)} (q_1, q_1') \xrightarrow{(s_2,s_2)} (q_2, q_2') : \text{final state}$$

Since s_1 and s_2 are labels, we can easily create a hedge of length 2 whose first tree is labeled s_1 and whose second tree is labeled s_2. This transition sequence says that, by reading such a hedge $\{s_1, s_2\}$, M will be in q_2, M' will be in q_2', and both are in the final state. So L_1 and L_2 are in fact in the ambiguous relation.

Let's look at the another example.

$$(q0, q0') \xrightarrow{(s_1,s_1')} (q_1, q_1') \xrightarrow{(s_2,s_2')} (q_2, q_2') : \text{final state}$$

This transition is possible only when (s_1, s_1') and (s_2, s_2') are both member of S^*. Since (s_1, s_1') is a member of S^*, s_1 and s_1' must be in the ambiguous relation. So does (s_2, s_2'). By definition of label ambiguity, there exists a nice XML element that can be interpreted by both s_1 and s_1'. Things are the same for s_2 and s_2'.

As a whole, we now know that there actually exists such a nice hedge that can be interpreted by both $\{s_1, s_2\}$ and $\{s_1', s_2'\}$. This hedge can be accepted by both M and M', therefore L_1 and L_2 are proven to be in the ambiguous relation.

So if M^* accepts something, it means there is a XML hedge that can be interpreted as both L_1 and L_2. Therefore those two labels are in the ambiguous relation.

Next, let's assume that L_1 and L_2 are in fact in the ambiguous relation and see if dTLA can detect this. From the definition of label ambiguity, there exists a hedge (let it be `<c1>...</c1><c2>...</c2> ...`) and there exists two interpretations I and I' such that

$$\begin{aligned} & \{I(c_1), I(c_2), I(c_3), \dots\} \text{ is accepted by } M \\ \wedge \quad & \{I'(c_1), I'(c_2), I'(c_3), \dots\} \text{ is accepted by } M' \end{aligned}$$

$I(c_1)$ means a label assigned to c_1 by I. When we "zip" the above two sequences, we have $\{\big(I(c_1), I'(c_1)\big), \big(I(c_2), I'(c_2)\big), \cdots\}$ and this sequence is always accepted by M^+ since M^+ is defined to treat all unknown relation as ambiguous.

So, in short, now I proved that if L_1 and L_2 are in fact in the ambiguous relation then M^+ always accepts something. In other words, if M^+ doesn't accept anything then L_1 and L_2 are in fact NOT in ambiguous relation.

In summary, I proved the followings.

1. If M^* accepts something, then L_1 and L_2 are in fact in the ambiguous relation.
2. If M^+ accepts nothing, then L_1 and L_2 are in fact NOT in the ambiguous relation.

Therefore, dTLA is proven to be always correct.

6 Algorithm to detect a catalyst

In this section, the algorithm that detects whether a label is a catalyst is proposed. This algorithm is named "detectCatalyst" (dC in short). First, I'll define catalyst in more formal way. this definition complies the requirement of catalyst: namely, "the grammar is ambiguous if and only if it has at least one catalyst" (Proposition 3.4.) Then I'll propose an algorithm to detect a catalyst. Finally, informal proof of correctness is provided.

6.1 Formal definition of catalyst

We have described a catalyst as a label that provides "a choice" between ambigous labels. Formally, label L is a catalyst if and only if

1. There exists two rules in the form of $L \rightarrow \mathbf{a} < C_1 >$ and $L \rightarrow \mathbf{a} < C_2 >$
2. There are two *different* sequences of labels $\{a_1, a_2, \ldots, a_n\}$ and $\{b_1, b_2, \ldots, b_n\}$ where either $a_i = b_i$ or $a_i \sim b_i$ (a_i and b_i are in the ambiguous relation).
3. And C_1 accepts the first sequence and C_2 accepts the second sequence.

Note that since they are two different sequences, we have at least one pair such that $a_i \neq b_i$ (thus $a_i \sim b_i$, they are in the ambiguous relation). The proof is given in the appendix.

6.2 Algorithm

Algorithm 3 dC

Input: L : the label which is going to be tested
Input: 2 A : a score matrix fully completed
  for all rule $R_i : L \rightarrow \mathbf{c} < M >$ whose label is L **do**
    for all rule $R_j : L \rightarrow \mathbf{c'} < M' >$ whose label is L **do**
      if c = c' **then**
        Let the content model automaton of R_i be (Q, S, T, q_0, F)
        Let the content model automaton of R_j be (Q', S, T', q'_0, F')
        $S^* \Leftarrow \left\{ (s, s') \middle| s \in S, s' \in S, (s = s' \vee s \sim s') \right\}$
        $T^*((q, q'), (s, s')) = (T(q, s), T'(q', s'))$
        $M^* = (Q \times Q', S^*, T^*, (q_0, q'_0), F \times F')$ { create a new automaton }
        remove all right-unreachable states from M^*.

        if M^* has a transition by (a, b) such that $a \sim b$ **then**
          output "L is a catalyst"
          exit function
        end if
      end if
    end for
  end for
  output "L is not a catalyst"

We know that all fields of the score matrix are fully marked.
dC produces one of the following output and always stops.

1. Label L is a catalyst
2. Label L is not a catalyst

Intuitively, this algorithm finds the "two sequences" that characterize the formal definition of catalyst.

This algorithm uses two **for** loop to test all possible combination of rules, because there can be multiple rules that share the same label.

To understand dC better, please write this M^* down to a big whiteboard. When writing a edge of (s, s), use a black pen. When writing a edge of the form of (s, s'), please use a red pen. This algorithm says "L is a catalyst" if there exists a path from (q_0, q_0) to a final state that uses at least one red edge. If no such path exists, the algorithm says "L is not a catalyst" and stops.

There are many ways to detect the existence of such a path. One way is, as used in dC, remove all right-unreachable states and see if a red edge still exists. Right-unreachable states are those states which have no path to the final state. Another way might be to use dijkstra-like algorithm to mark all states.

6.3 How it works

Say dC said "L is a catalyst" and stopped. This occurs only if there exists a sequence or label pair

$$\Big\{(a_1, b_1), (a_2, b_2), (a_3, b_3), \dots\Big\}$$

And this sequence is accepted by M^*. We know that at least one of the pair is a "red edge". That is $a_i \neq b_i$ for a certain i but a_i and b_i are in the ambiguous relation.

By "unzipping" this sequence, we obtain two sequences $\{a_1, a_2, \dots\}$ and $\{b_1, b_2, \dots\}$ of the same length.

From the way I defined T^*, we can say that both sequences can be accepted by M and M' respectively. So these two sequences are the "two sequences" that characterizes L being in fact a catalyst.

Thus soundness is proved: if dC says "it's a catalyst", then it actually is.

Next, I'll prove that if it is actually a catalyst, the algorithm always says "it's a catalyst" (completeness).

If L is actually a catalyst, there exists the two sequences $\{a_1, a_2, \dots\}$ and $\{b_1, b_2, \dots\}$ (the formar is accepted by M and the latter is accepted by M'). Every pair (a_i, b_i) is either equal or in the ambiguous relation. And at least one pair is not equal. Furthermore, both of the sequences can be accepted by M by the following transition sequences.

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \to \cdots \to q_f$$
$$q_0 \xrightarrow{b_1} q'_1 \xrightarrow{b_2} q'_2 \to \cdots \to q'_f$$

Since both are accepted by M and M' respectively, both q_f and q'_f are final states. Let's "zip" those two sequences into one sequence.

$$\big\{(a_1, b_1), (a_2, b_2), \dots\big\}$$

What happens to M^* if it receives this sequence? The transition sequence of M^* will be as follows:

$$(q_0, q_0) \xrightarrow{(s_1, s'_1)} (q_1, q'_1) \xrightarrow{(s2, s2')} (q_2, q'_2) \to \cdots \to (q_f, q'_f)$$

As I mentioned earlier, both q_f and q'_f are final states. So (q_f, q'_f) is the final state of M^*, which means this sequence is accepted by M^*. And the fact that at least one pair (a_i, b_i) is not equal guarantees that this transition uses a red edge for this pair. So, in cases like this, the algorithm always stops by saying "L is a catalyst".

This proves completeness: if it is actually a catalyst, the algorithm always detects that.

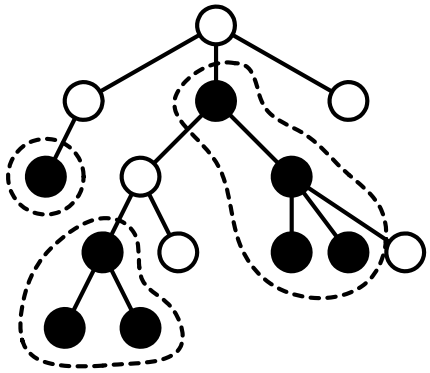
7 Completeness of the entire algorithm

In this section, We will have a closer look to blanks in the score matrix left after the iterative execution of dTLA. In the outline of the algorithm, I wrote that those blanks can safely be assumed as unambiguous. But why? this section answers this question by proving the following proposition.

Proposition 7.1 If blanks are left in the score matrix after the iteration, those label pairs are not in the ambiguous relation.

Let G be an ambiguous grammar, and X be the XML instance which can be interpreted in two ways (I and I'). We know the existence of such X, but don't know what X actually is. Since I and I' are different, there must exists elements that are given a different labels by I and I' (solid circles in the figure below). Such elements can be regarded as a set of trees (as enclosed in dotted line in the figure).

Let T be one such tree. All we know about T is that it exists, its size is finite, and its elements are given a different interpretation by I and I'.



Consider a root node of T. There must be such a node, although all we know is its existence. By definition of T, this node has two different labels. Let they be L_{r1} and L_{r2}. We don't know what L_{r1} and L_{r2} actually are, but we do know that those labels are in the grammar G, and $L_{r1} \neq L_{r2}$.

What I'd like to prove is, the score matrix always have 'A'mark for these L_{r1} and L_{r2}: that is, dTLA algorithm always finds that they are in the ambiguous relation. (Note that we never know what L_{r1} and L_{r2} actually are.)

Let us go back to the tree T, of which every node has two different interpretations. Because the size of T is finite, we always have at least one "leaf", which means none of its children is a member of T. Consider the set of all such nodes and name it N_0. N_0 is not empty. Let $n \in N_0$. n has two different labels, so name it $I(n)$ and $I'(n)$ respectively. Let $c_1, c_2, \ldots c_i$ be children of n (i may be 0). Due to the way we choose n, all have the same interpretation under I and I'.

Although we don't know what $I(n)$ and $I'(n)$ actually are, but because dTLA tests every possible pair of two labels, dTLA algorithm must have tested $I(n)$ and $I'(n)$ in each iteration.

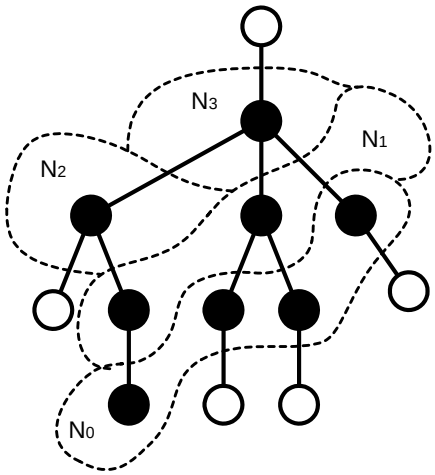
dTLA is capable of detecting ambiguity of $I(n)$ and $I'(n)$ even if the score matrix is fully blank. Intuitively, the fact that each children of n has only one label, while n itself has two different labels, are so decisive that dTLA cannot miss this situation.

In summary, although we don't know actual members of N_0, we know that for every $n \in N_0$, the score matrix always has 'A'mark for $I(n)$ and $I'(n)$ after the first iteration.

So if the height of all such possible T is 1, proposition 7.1 is proved.

Of course, there may be a tree T of height 2 or more. If the height of T is 2 or more, we always have at least one node that satisfies the following requirement.

Every child of n is either (1) a member of N_0, or (2) it has only one label.



There may be more than one nodes that satisfy this requirement. But there always at least one such node since we assume height of T is more than or equal to 2. Consider a set of all such nodes and name it N_1.

Again, we don't know the actual members of N_1, but it exists. So pick one node from N_1 and name it n.

$I(n)$ is a label of n by the interpretaion I, and $I'(n)$ is a label by the interpretation I'. Apparently, they are the members of the grammar G. Therefore, since dTLA tests every pair, this $(I(n), I'(n))$ pair is also tested.

And again, dTLA algorithm is wise enough to decide these two labels are in the ambiguous relation only if one condition is met. That condition is, for every $n' \in N_0$, we have 'A'mark for $I(n')$ and $I'(n')$.

As we proved ealier, after the first iteration, this condition is met. Therefore in the second iteration, dTLA detects that $I(n)$ and $I'(n)$ are in the ambigous relation. Because of the soundness of dTLA algorithm, it never judges $I(n)$ and $I'(n)$ are not in the ambigous relation. Therefore in the first iteration, $I(n)$ and $I'(n)$ are judged as either:

1. their relation is undecidable at this moment, or
2. they are in the ambiguous relation.

If the former is the case, dTLA detects that they are in the ambigous relation in the second iteration. If the latter is the case, that is also fine. As a whole, For every T of height 2 or less, then the score matrix always has 'A'mark for L_{r1} and L_{r2}, after the second iteration.

In general, when the height of T is h or more, we can predict the existence of n_{h-1} such that

every child c of n_{h-1} is either

- a member of N_0, or
- a member of N_1, or
- $\cdots$
- a member of N_{h-2}, or
- it has only one label ($I(c) = I'(c)$)

By using the same reasoning as above, dTLA detects the ambiguous relation of $I(n_{h-1})$ and $I'(n_{h-1})$ in its $(h-1)$th iteration. For any T, its height is always finite. Therefore by the induction on the height of tree, it is proved that dTLA can detect all ambiguous labels.

For any X that has more than one interpretations, its T is always finite in its height. So proposition 7.1 is proved.

8 Conclusion

An algorithm to detect ambiguity of regular tree grammar (e.g., RELAX and TREX) is presented. This algorithm is sound as well as complete.

DTD has long been the only practical schema language for XML. Since any schemata that can be written in DTD is guaranteed to be unambiguous, ambiguity of schemata was not a problem until recently. However, in the era of modern schema languages, ambiguous grammar affects various schema processings like validation, schema inference, and type-assignment.

With regard to this algorithm, one important question is whether "inherently ambiguous language" exists or not. That is, a language such that any attempt to *naturally* capture the language ends up in ambiguous grammars.

Several things are left unsolved. First, we should extend "clause" to capture attribute constraints. Unfortunately, powerful datatyping capabilities that are seen in XML Schema will probably break completeness of this algorithm. Second, we should also cover typed string in the content model. The algorithm will be almost identical, but it is also likely to break completeness.

9 Acknowledgement

I would like to express my sincere gratitude to Murali Mani for his insightful commnts. I would also like to thank Makoto MURATA for his comments.

References

[1] RELAX Core (REgular LAnguage description for Xml, JIS-TR 0029:2000)
`http://www.xml.gr.jp/relax/`

[2] TREX (Tree REgular expressions for Xml)
`http://www.oasis-open.org/ committees/trex/index.shtml`

[3] XML Schema, W3C Proposed Recommendation, Mar. 2001. `http://www.w3.org/TR/xmlschema-1/`

[4] Makoto Murata, Dongwon Lee, and Murali Mani. "Taxonomy of XML Schema Languages using Formal Language Theory", 2000. `http://www.cs.ucla.edu/~mani/xml/mura0106.ps`

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

10 Appendix: Proof of well-definedness of catalyst

the grammar is ambiguous if and only if it has at least one catalyst.

To prove that the formal definition complies the above requirement of catalyst, following two proofs are necessary and suffice:

- If there exists a label L that satisfied our definition of catalyst, then the grammar is ambiguous.
- If the grammar is ambiguous, there exists a label L that satisfies our definition of catalyst.

Say label L is a catalyst in the sense of the formal definition of catalyst. By that definition, we know that there actually exists two such sequences.

For every pair (a_i, b_i), there exists a tree whose top element can be interpreted by both a_i and b_i (if $a_i = b_i$ then this tree is trivial. if $a_i \neq b_i$ then $a_i \sim b_i$ and the existence is guaranteed by the ambiguous relation).

By concatenating those trees into a hedge, we can say that there exists a hedge that can be interpreted by both sequences. Again by the formal definition of catalyst, label L accepts both interpretations. So now we are succeeded in creating the tree whose top element is L and which has two different interpretations. Therefore, trivially the entire grammar becomes ambiguous.

The above reasoning proves the first part.

The rest is to prove that if a grammar is ambiguous, there exists a catalyst in it.

Assume G is an ambiguous grammar. By the definition of ambiguous grammar, we can assign two different interpretation (I and I') to G. Since I and I' are different, there must be a node n such that $I(n) \neq I'(n)$.

Without a loss of generality, we can assume that the parent node of n has assigned the same label L by two interpretations. (If $I(p(n)) \neq I'(p(n))$, then we can use $p(n)$ as n because tree is finite and the root element has always label S.)

Let the parent node of n be $p(n)$, and its children be $c_1, \ldots c_m$. Consider the two sequences $\{I(c_1), \ldots, I(c_m)\}$ and $\{I'(c_1), \ldots, I'(c_m)\}$. Each $I(c_i)$ and $I'(c_i)$ is either equal or in the ambiguous relation, and at least one pair is not equal because n is one of $c_1, \ldots, c_m$ and $I(n) \neq I'(n)$.

These two sequences satisfies our formal definition of catalyst. Therefore, L is a catalyst.

Q.E.D.