# Assisting the Design of XML Schema: Diagnosing Nondeterministic Content Models

Haiming Chen, Ping Lu

State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing 100190, China
{chm,luping}@ios.ac.cn

**Abstract.** One difficulty in the design of XML Schema is the restriction that the content models should be deterministic, i. e., the Unique Particle Attribution (UPA) constraint, which means that the content models are deterministic regular expressions. This determinism is defined semantically without known syntactic definition for it, thus making it difficult for users to design. Presently however, no work can provide diagnostic information if content models are nondeterministic, although this will be of great help for designers to understand and modify nondeterministic ones. In the paper we investigate algorithms that check if a regular expression is deterministic and provide diagnostic information if the expression is not deterministic. With the information provided by the algorithms, designers will be clearer about why an expression is not deterministic. Thus it contributes to reduce the difficulty of designing XML Schema.
**keywords**: XML Schema, deterministic content models, diagnostic information

## 1 Introduction

Extensible Markup Language (XML) has become popular for the Web and other applications. Usually in applications XML data are provided with schemas that the XML data must conform to. These schemas are important for solving problems and improving efficiency in many tasks of XML processing, for example, in query processing, data integration, typechecking, and so on. Among the many schema languages for XML, XML Schema which is recommended by W3C has been the most commonly used one. It is not easy, however, to design a correct XML Schema: investigation reveals that many XML Schema Definitions in practice have errors [3, 7]. One difficulty in designing XML Schema is the restriction that the content models should be deterministic, i. e., the Unique Particle Attribution (UPA) constraint, which means that the content models are deterministic regular expressions. In another XML schema language recommended by W3C, Document Type Definition (DTD), deterministic content models are also used.

A regular expression is deterministic (or one-unambiguous) if, informally, a symbol in the input word should be matched uniquely to a position in the regular expression without looking ahead in the word. This determinism, however, is

defined semantically without known syntactic definition for it, thus making it difficult for users to design.

Brüggemann-Klein [4] showed that deterministic regular expressions are characterized by deterministic Glushkov automata, and whether an expression is deterministic can be decided. In [5] an algorithm is provided to decide whether a regular language, given by an arbitrary regular expression, is deterministic, i. e., can be represented by a deterministic regular expression. An algorithm is further given there to construct equivalent deterministic regular expressions for nondeterministic expressions when they define deterministic languages. The size of the constructed deterministic regular expressions, however, can be exponentially larger than the original regular expressions. In [1] an algorithm is proposed to construct approximate deterministic regular expressions for regular languages that are not deterministic. Bex *et al.* [2] further provided improved algorithms for constructing deterministic regular expressions or constructing approximations.

All existing work, however, cannot provide diagnostic information of nondeterministic expressions. Consider if a design tool can locate the error positions and tell the type of error making the expression nondeterministic, just like what compilers or other program analysis tools do for programs, then it will be greatly helpful for designers to understand and modify nondeterministic expressions. Note here *error* is used to denote what make an expression nondeterministic.

In this paper we tackle the above issue. Our aim is to provide as much diagnostic information for errors as possible when expressions are nondeterministic. The idea is, if we can check expressions at the syntactic level, then it is easier to locate errors. Following [5], we designed a conservative algorithm, which will accept deterministic expressions, but may also reject some deterministic expressions. We improved the conservative algorithm by borrowing semantic processing and obtained an exact checking algorithm, which will accept all deterministic expressions, and reject only nondeterministic expressions. But it will not provide as precise diagnostic information for some nondeterministic expressions as for the other nondeterministic expressions. We further presented a sufficient and necessary condition for deterministic expressions, which leads to another exact checking algorithm for deterministic expressions. With the information provided by the algorithms, designers will be clearer about why an expression is not deterministic. Thus the difficulty of designing deterministic expressions, and, of designing XML Schema at large, is lowered.

We also implemented the algorithm in [5] which constructs an equivalent deterministic expression if an expression is not deterministic but defines a deterministic language, as an alternative way to obtain deterministic expressions.

We conducted several preliminary experiments, and the experimental results are presented. The main contributions of the paper are as follows.

(1) We propose the notion of diagnosing deterministic regular expressions. While similar notion has been familiar in other areas of software, this notion is missing for deterministic regular expressions due to the semantic nature.

(2) We prove several properties of deterministic regular expressions, which are the base of the algorithms.

(3) We present several algorithms for checking deterministic regular expressions and providing diagnostic information. The algorithms presented in the paper check if regular expressions are deterministic at the syntactic level of the expressions. The first algorithm uses stronger but syntactical conditions and is conservative. The second algorithm is exact, but may not obtain as precise diagnostic information for some regular expressions as for the other regular expressions. The third algorithm is also exact, and may obtain diagnostic information for all nondeterministic regular expressions. The work of the paper can be considered as a first step towards syntactic solutions of detecting deterministic regular expressions.

The above algorithms can be used in schema design tools in which designers are able to find and fix bugs iteratively.

There is another issue that is connected with the present issue. That is, since deterministic regular expressions denote a proper subclass of regular languages, if a nondeterministic expression does not define a deterministic language, then the expression cannot have any equivalent deterministic expression. So when an expression is nondeterministic it is useful to tell the designer in the mean time whether the expression denotes a deterministic language.

Section 2 introduces preliminary definitions. The algorithms are presented in Section 3, with a discussion of diagnostic information and examples. Experiments are presented in Section 4. Section 5 contains a conclusion.

## 2 Preliminaries

Let $\Sigma$ be an alphabet of symbols. The set of all finite words over $\Sigma$ is denoted by $\Sigma^*$. The empty word is denoted by $\varepsilon$. A regular expression over $\Sigma$ is $\emptyset, \varepsilon$ or $a \in \Sigma$, or is the union $E_1 + E_2$, the concatenation $E_1 E_2$, or the star $E_1^*$ for regular expressions $E_1$ and $E_2$. For a regular expression $E$, the language specified by $E$ is denoted by $L(E)$. Define $EPT(E) = true$ if $\varepsilon \in L(E)$ and $false$ otherwise. The symbols that occur in $E$, which is the smallest alphabet of $E$, is denoted by $\Sigma_E$.

For a regular expression we can mark symbols with subscripts so that in the marked expression each marked symbol occurs only once. For example $(a_1 + b_2)^* a_3 b_4 (a_5 + b_6)$ is a marking of the expression $(a + b)^* ab(a + b)$. The marking of an expression $E$ is denoted by $\overline{E}$. The same notation will also be used for dropping of subscripts from the marked symbols: $\overline{\overline{E}} = E$. The subscribed symbols are called *positions* of the expression. We extend the notation for words and automata in the obvious way. It will be clear from the context whether $\bar{\cdot}$ adds or drops subscripts.

**Definition 1.** *An expression $E$ is deterministic if and only if, for all words $uxv, uyw \in L(\overline{E})$ where $|x| = |y| = 1$, if $x \neq y$ then $\overline{x} \neq \overline{y}$. A regular language is deterministic if it is denoted by some deterministic expression.*

For an expression $E$ over $\Sigma$, we define the following functions:

$$first(E) = \{a \mid aw \in L(E), a \in \Sigma, w \in \Sigma^*\}$$
$$last(E) = \{a \mid wa \in L(E), w \in \Sigma^*, a \in \Sigma\}$$
$$follow(E, a) = \{b \mid uabv \in L(E), u, v \in \Sigma^*, b \in \Sigma\}, \text{for } a \in \Sigma$$

One can easily write equivalent inductive definitions of the above functions on $E$, which is omitted here.

Define $followlast(E) = \{b \mid vbw \in L(E), v \in L(E), v \neq \varepsilon, b \in \Sigma, w \in \Sigma^*\}$. An expression $E$ is in *star normal form* (SNF) [4] if, for each starred subexpression $H^*$ of $E$, $followlast(\overline{H}) \cap first(\overline{H}) = \emptyset$ and $\varepsilon \notin L(H)$.

A finite automaton is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where $Q$ is a finite set of states, $\Sigma$ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition mapping, $q_0$ is the start state, and $F \subseteq Q$ is the set of accepting states. Denote the language accepted by the automaton $M$ by $L(M)$.

## 3 Determining and diagnosing nondeterministic expressions

### 3.1 Algorithms

The Glushkov automaton was introduced independently by Glushkov [6] and McNaughton and Yamada [8]. It is known that deterministic regular expressions can be characterized by Glushkov automata.

**Lemma 1.** ([5]) *A regular expression is deterministic if and only if its Glushkov automaton is deterministic.*

Lemma 1 has led to an algorithm to check if an expression is deterministic [4]. We call this algorithm the semantic checking algorithm in the paper.

If diagnostic information about why a regular expression is not deterministic is required, we need a syntactic characterization, or at least some syntactic properties, of deterministic expressions. Such a characterization, however, is not known presently. We started from a property of deterministic expressions in star normal form in [5] by modifying it to more general expressions. To do this, Lemma 2 is required.

**Lemma 2.** *For a regular expression $E$, if $followlast(E) \cap first(E) = \emptyset$ then $followlast(\overline{E}) \cap first(\overline{E}) = \emptyset$.*

*Proof.* Suppose $x \in followlast(\overline{E}) \cap first(\overline{E})$, then $\overline{x} \in \overline{followlast(\overline{E})} = followlast(E)$ and $\overline{x} \in \overline{first(\overline{E})} = first(E)$, which is a contradiction. □

The following is a modified version of the afore mentioned property proved in [5][1].

[1] The original property in [5] requires the expression to be in star normal form. It was mentioned in the proof that this condition can be removed with slight change of the property.

**Lemma 3.** *Let $E$ be a regular expression.*
*$E = \emptyset, E = \varepsilon$, or $E = a \in \Sigma$: $E$ is deterministic.*
*$E = E_1 + E_2$: $E$ is deterministic iff $E_1$ and $E_2$ are deterministic and $first(E_1) \cap first(E_2) = \emptyset$.*
*$E = E_1E_2$: If $L(E) = \emptyset$, then $E$ is deterministic. If $L(E) \neq \emptyset$ and $\varepsilon \in L(E_1)$, then $E$ is deterministic iff $E_1$ and $E_2$ are deterministic, $first(E_1) \cap first(E_2) = \emptyset$, and $followlast(E_1) \cap first(E_2) = \emptyset$. If $L(E) \neq \emptyset$ and $\varepsilon \notin L(E_1)$, then $E$ is deterministic iff $E_1$ and $E_2$ are deterministic and $followlast(E_1) \cap first(E_2) = \emptyset$.*
*$E = E_1^*$: $E$ is deterministic and $followlast(\overline{E_1}) \cap first(\overline{E_1}) = \emptyset$ iff $E_1$ is deterministic and $followlast(E_1) \cap first(E_1) = \emptyset$.*

The last case of Lemma 3 can be proved from a modification of the proof in [5] in addition with Lemma 2. The proof of the other cases is the same as the proof in [5].

This property, however, is not a sufficient and necessary condition of deterministic expressions, since in the last case the expression $E$ is accompanied with an additional condition. Actually, there are examples in which either $E$ is deterministic and $followlast(E_1) \cap first(E_1) \neq \emptyset$, or $E_1$ is deterministic but $E$ is not deterministic. In other words, the condition that $E_1$ is deterministic and $followlast(E_1) \cap first(E_1) = \emptyset$ is too strong to ensure $E_1^*$ to be deterministic.

**Proposition 1.** *For a regular expression $E = E_1^*$,*
*(1) $E$ can be deterministic when $followlast(E_1) \cap first(E_1) \neq \emptyset$.*
*(2) If $E$ is deterministic then $E_1$ is deterministic, but not vice versa.*

*Proof.* (1) This can be shown by an example: $E = (a^*)^*$, where $E$ is deterministic because it contains only one symbol, but $followlast(a^*) \cap first(a^*) = \{a\}$.

(2) ($\Rightarrow$): This is obvious. If $E_1$ is not deterministic, then $E$ cannot be deterministic, since any word of $\overline{E_1}$ is also a word of $\overline{E}$, which in turn assures the following: a pair of words violating the deterministic condition of $E_1$ will also violate the deterministic condition of $E$.

The reverse can be shown by an example: $(a(a + \varepsilon))^*$. One can verify that $a(a + \varepsilon)$ is deterministic, but $(a(a + \varepsilon))^*$ is not. □

On the other hand, up to date there is no known simpler condition for the last case of Lemma 3.

In order to check if an expression is deterministic and locate error position when the expression is not deterministic, one way is to directly use the property of Lemma 3, thus resulting in a conservative algorithm, i. e., if it accepts an expression, then the expression must be deterministic, but it may also reject some deterministic expressions.

To obtain an exact checking algorithm we make some compromise and use the following strategies, based on the above properties. When $e = e_1^*$ we first check if $e_1$ is deterministic using Lemma 3. If $e_1$ is not deterministic, then $e$ is not either by Proposition 1. Furthermore, if the erroneous part in $e_1$ is not a starred subexpression, then precise diagnostic information can be obtained. If $e_1$ is deterministic, and $followlast(e_1) \cap first(e_1) = \emptyset$, then $e$ is deterministic by

Lemma 3. Otherwise, we encounter the only uncertain case: $e_1$ is deterministic and $followlast(e_1) \cap first(e_1) \neq \emptyset$, and shift to semantic level and use the semantic checking algorithm [4] to check if $e$ is deterministic. If $e$ is deterministic, then the algorithm proceeds smoothly without any impact of the semantic checking. If $e$ is not deterministic, then we can only say that $e$ is nondeterministic to the users, without any further diagnostic information. The resulting algorithm is exact: it will accept all deterministic expressions, and reject only nondeterministic expressions. Notice that the semantic checking algorithm runs in linear time [4], so will not much lower down the efficiency of the whole algorithm. The cost is, it will not provide as precise diagnostic information for some nondeterministic expressions as for the other nondeterministic expressions.

Further, for any expression $E = E_1^*$, we give a sufficient and necessary condition of $E$ being deterministic as follows.

**Proposition 2.** *For $E = E_1^*$, $E$ is deterministic iff $E_1$ is deterministic and $\forall y_1 \in followlast(\overline{E_1}), \forall y_2 \in first(\overline{E_1})$, if $\overline{y_1} = \overline{y_2}$ then $y_1 = y_2$.*

By Proposition 2 and using Lemma 3 we get a sufficient and necessary condition for deterministic expressions. This gives another algorithm which provide diagnostic information for all nondeterministic expressions.

The above proposition actually requires that if $followlast(E_1)$ and $first(E_1)$ have common elements, the intersection can only be in the same positions of $E_1$. To ease the computation and obtain more diagnostic information, we present the following inductive computation of the condition.

**Definition 2.** *The function $\mathcal{P}(E)$ which returns true or false is defined as*

$$\begin{aligned}
&\mathcal{P}(\varepsilon) = \mathcal{P}(\emptyset) = \mathcal{P}(a) = true \quad a \in \Sigma \\
&\mathcal{P}(E_1 + E_2) = \mathcal{P}(E_1) \wedge \mathcal{P}(E_2) \wedge (followlast(E_2) \cap fitst(E_1) = \emptyset) \\
&\quad \wedge (followlast(E_1) \cap fitst(E_2) = \emptyset) \\
&\mathcal{P}(E_1E_2) = (\neg(EPT(E_1)) \wedge \neg(EPT(E_2)) \wedge (followlast(E_2) \cap fitst(E_1) = \emptyset)) \vee \\
&\quad (EPT(E_1) \wedge \neg(EPT(E_2)) \wedge \mathcal{P}(E_2) \wedge (followlast(E_2) \cap fitst(E_1) = \emptyset)) \vee \\
&\quad (\neg(EPT(E_1)) \wedge EPT(E_2) \wedge \mathcal{P}(E_1) \wedge (followlast(E_2) \cap fitst(E_1) = \emptyset)) \vee \\
&\quad (EPT(E_1) \wedge EPT(E_2) \wedge \mathcal{P}(E_1) \wedge \mathcal{P}(E_2) \wedge (followlast(E_2) \cap fitst(E_1) = \emptyset)) \\
&\mathcal{P}(E_1^*) = \mathcal{P}(E_1)
\end{aligned}$$

**Proposition 3.** *Let an expression $E$ be deterministic. The following two statements are equivalent.*
*(1) $\forall y_1 \in followlast(\overline{E}), \forall y_2 \in first(\overline{E})$, if $\overline{y_1} = \overline{y_2}$ then $y_1 = y_2$.*
*(2) $\mathcal{P}(E) = true$.*

In the following we give the three algorithms. The conservative one is *deterministic_c*, the second one is *deterministic*, and the third one is *deterministicpl*.

**Algorithm 1** *deterministic_c*($e$): Boolean
**Input**: a regular expression $e$
**Output**: true if $e$ is deterministic or false and diagnostic information otherwise

1. **if** $e = \emptyset, \varepsilon$, or $a$ for $a \in \Sigma$ **then return** true

2. **if** $e = e_1 + e_2$ **then**
3. **if** *deterministic_c*($e_1$) and *deterministic_c*($e_2$) **then**
4. **if** $first(e_1) \cap first(e_2) = \emptyset$ **then return** true **else**
5. {*print_err*; **return** false}
6. **else return** false
7. **if** $e = e_1 e_2$ **then**
8. **if** *deterministic_c*($e_1$) and *deterministic_c*($e_2$) **then**
9. **if** $followlast(e_1) \cap first(e_2) \neq \emptyset$ **then** {*print_err*; **return** false} **else**
10. **if** *lambda*($e_1$) **then**
11. **if** $first(e_1) \cap first(e_2) \neq \emptyset$ **then** {*print_err*; **return** false} **else**
12. **return** true
13. **else return** true
14. **else return** false
15. **if** $e = e_1^*$ **then**
16. **if** *deterministic_c*($e_1$) **then**
17. **if** $followlast(e_1) \cap first(e_1) = \emptyset$ **then return** true **else**
18. {*print_err*; **return** false}
19. **else return** false

**Algorithm 2** *deterministic*($e$): Boolean
**Input**: a regular expression $e$
**Output**: true if $e$ is deterministic or false and diagnostic information otherwise
(1 – 14 are the same as *deterministic_c*)

15. **if** $e = e_1^*$ **then**
16. **if not** *deterministic*($e_1$) **then return** false
17. **if** $followlast(e_1) \cap first(e_1) = \emptyset$ **then return** true **else**
18. **if** *isdtre*($e$) **then return** true **else**
19. {*print_err*; **return** false}

**Algorithm 3** *deterministicpl*($e$): Boolean
**Input**: a regular expression $e$
**Output**: true if $e$ is deterministic or false and diagnostic information otherwise
(1 – 14 are the same as *deterministic_c*)

15. **if** $e = e_1^*$ **then**
16. **if not** *deterministicpl*($e_1$) **then return** false
17. **if** $P(e_1)$ **then return** true **else**
18. **return** false

All of the algorithms take as input a regular expression, and output a Boolean value indicating if the expression is deterministic as well as diagnostic information if the expression is not deterministic. In the algorithm *determinstc_c*, *lambda*($e$) is just the function $EPT(e)$ which returns true if $\varepsilon \in L(e)$ and false otherwise. *print_err* is not a real function here, it just indicates some statements in the implementation that print current error information. For example, in line 5 *print_err* should print that $first(e_1) \cap first(e_2)$ is not empty, and in line 9 *print_err* should print that $followlast(e_1) \cap first(e_2)$ is not empty. It is not

difficult to indicate the positions of $e_1$ and $e_2$ by the parse tree of the whole expression. The difference between *deterministic_c* and the other algorithms only starts from line 16. In *deterministic*, *isdtre* is the semantic checking algorithm [4] to check if a regular expression is deterministic. It is used in the case of $e = e_1^*$. In *deterministicpl*, $P(e)$ calculates $\mathcal{P}(e)$ and print diagnostic information if $e$ is nondeterministic.

**Theorem 1.** *If deterministic_c(e) returns true, then e is deterministic.*

*Proof.* It follows directly from Lemma 3. □

**Theorem 2.** *deterministic(e) returns true if and only if e is deterministic.*

*Proof.* It follows from Lemma 3, Lemma 1, and Proposition 1. □

**Theorem 3.** *deterministicpl(e) returns true if and only if e is deterministic.*

*Proof.* It follows from Lemma 3, Proposition 2, and Proposition 3. □

### 3.2 Reporting errors

Three kinds of error information can be reported by the above algorithms:

- Error location. Using the parse tree of an expression, the subexpressions that cause an error can be located precisely.

- Types of errors. There are roughly the following types of errors:

(1) $first$-$first$ error, indicating $first(e_1) \cap first(e_2) \neq \emptyset$. It can further be classified into $first$-$first$-+ and $first$-$first$-., corresponding to a (sub)expression $e = e_1 + e_2$ and $e = e_1 + e_2$ respectively.

(2) $followlast$-$first$ error, indicating $followlast(e_1) \cap first(e_2) \neq \emptyset$. Similarly it is also classified into $followlast$-$first$-+, $followlast$-$first$-., and $followlast$-$first$-*.

(3) A starred (sub)expression is not deterministic, indicating the semantic checking error in *deterministic*.

(4) $followlast$-$first$-nd error, indicating an error of $followlast(e_1) \cap first(e_2) \neq \emptyset$ in $\mathcal{P}(e)$, corresponding to a violation of the condition is Proposition 2.

- Other diagnostic information. For example, for a type (1) error, the two $first$ sets can be provided. For a type (2) error, besides the $followlast$ and $first$ sets, the symbols in the *last* set that cause the overlap of the $followlast$ and $first$ sets, and symbols in the $follow$ set of the previous symbols, can be provided.

In addition to the above information, other information like parse trees of expressions, the Glushkov automata, and the matching positions of a word against an expression can also be displayed, thus providing debugging facilities of expressions.

Of course, in an implementation of a tool, the above information can be implemented such that the users can select which information to display.

### 3.3 Examples

*Example 1.* Suppose one want to write a schema for papers with no more than two authors. The content model of the papers can be defined as

`Title, Author?, Author, Date, Abstract, Text, References` [2]

which equals to the following regular expression

`Title, (Author+empty), Author, Date, Abstract, Text, References`

By using the above algorithms, the following information is displayed[3]:

```
error: the expression is not deterministic.
error found in: "Title, (Author+empty), Author"
hints: the sets of followlast((Title, (Author+empty))) and
       first(Author) have common elem
       followlast((Title, (Author+empty)))={Author(2)}
         trace: Title in last((Title, (Author+empty)))
                follow((Title, (Author+empty)), Title)={Author}
       first(Author)={Author(3)}
```

Then the content model can be rewritten into the following:

`Title, Author, Author?, Date, Abstract, Text, References`

which is deterministic.

Of course for some nondeterministic content models their equivalent deterministic ones are very difficult to find, as in the following example.

*Example 2.* $(a + b)^*a$ defines any string of $a$ or $b$, including the empty word, followed by one $a$. The above algorithms will show that this expression is not deterministic, and this is because $EPT((a + b)^*)$ and $followlast((a + b)^*) \cap first(a) \neq \emptyset$ and $first((a + b)^*) \cap first(a) \neq \emptyset$. For the expression it is difficult to write an equivalent deterministic regular expression.

However, using the diagnostic information, the designer can change the design to circumvent the error: $(a + b)^*c$, and $c$ is defined as $a$ in another rule.

So the diagnostic information can
- help the designer to locate errors and rewrite nondeterministic expressions into correct ones, and
- help the designer to understand the reasons of errors, or change design to circumvent the errors.

## 4 Experiments

We have implemented the algorithms and performed some experiments. The algorithms were tested with generated regular expressions in different sizes. We use the number of symbol occurrences in the expression, or the alphabetic width

[2] The comma (`,`) denotes concatenation

[3] Concatenation and union are assumed to be left associative as usual in expressions

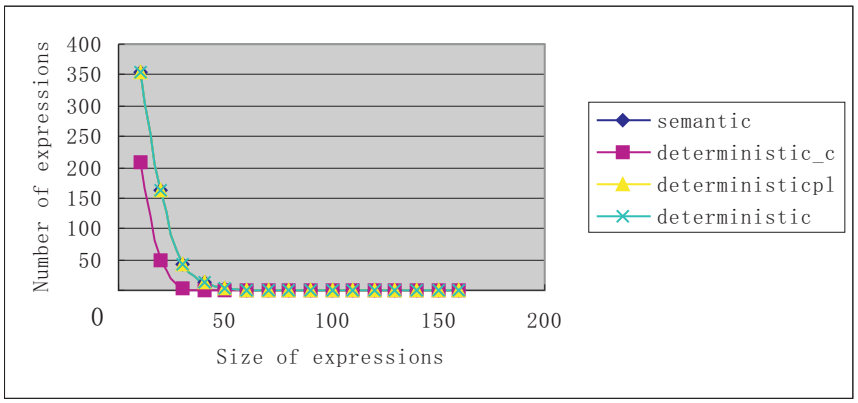


**Fig. 1.** Numbers of deterministic regular expressions

of the expression to denote the size of a regular expression. The sizes of regular expressions ranges from 10 to 160 in the experiments, with 500 expressions in each size. The size of the alphabet was set to 40. The algorithms were implemented in C++. The experiments were run on Intel core 2 Duo 2.8GHz, 4GB RAM.

Figure 1 shows the numbers of deterministic regular expressions determined by each algorithm, in which 'semantic' denotes the semantic checking algorithm, others are clear from their name. We can observe the number of deterministic expressions decreases when the size of regular expressions grows. Intuitively, this is because the possibility of the occurrences of the same symbol increase when the size of expressions increases, which increases the possibility of nondeterminism. Actually, when size is greater than 50, all expressions are nondeterministic. The numbers of deterministic regular expressions identified by the algorithm *deterministic_c* are less than the numbers identified by the other algorithms, which coincides with that *deterministic_c* is conservative. The numbers of deterministic expressions determined by the algorithms *deterministic* and *deterministicpl* are identical with the numbers determined by the semantic checking algorithm, reflecting that *deterministic* and *deterministicpl* exactly detect all deterministic expressions.

The errors in the tested regular expressions found by the algorithms are shown in Table 1. The first column in the table shows the size of regular expressions. The other columns include the numbers of different types of errors caught by *deterministic_c*, *deterministic* and *deterministicpl* in the experiment, in which `first-first`, `fola-first`, `star-exp`, and `ff-nd` correspond to the types (1), (2), (3), and (4) of errors presented in Section 3.2, respectively. It shows that the most common type of errors in the experiment is the *first-first* errors. Also the numbers of errors for each of `first-first` and `fola-first` of *deterministicpl* and *deterministic* are identical, and the numbers of errors for `ff-nd` and `star-exp` are identical too. This is because the two algorithms both exactly

**Table 1.** Errors found by the algorithms

| size | deterministic_c | | deterministicpl | | | deterministic | | |
|---|---|---|---|---|---|---|---|---|
| | first-first | fola-first | first-first | fola-first | ff-nd | first-first | fola-first | star-exp |
| 10 | 107 | 37 | 135 | 5 | 4 | 135 | 5 | 4 |
| 20 | 184 | 153 | 305 | 12 | 20 | 305 | 12 | 20 |
| 30 | 244 | 213 | 422 | 9 | 26 | 422 | 9 | 26 |
| 40 | 263 | 225 | 448 | 16 | 24 | 448 | 16 | 24 |
| 50 | 266 | 231 | 450 | 23 | 24 | 450 | 23 | 24 |
| 60 | 274 | 226 | 454 | 23 | 23 | 454 | 23 | 23 |
| 70 | 271 | 229 | 451 | 18 | 31 | 451 | 18 | 31 |
| 80 | 267 | 233 | 463 | 15 | 22 | 463 | 15 | 22 |
| 90 | 259 | 241 | 460 | 20 | 20 | 460 | 20 | 20 |
| 100 | 300 | 200 | 465 | 21 | 14 | 465 | 21 | 14 |
| 110 | 285 | 215 | 472 | 15 | 13 | 472 | 15 | 13 |
| 120 | 280 | 220 | 466 | 20 | 14 | 466 | 20 | 14 |
| 130 | 282 | 218 | 469 | 12 | 19 | 469 | 12 | 19 |
| 140 | 290 | 210 | 469 | 10 | 21 | 469 | 10 | 21 |
| 150 | 299 | 201 | 461 | 22 | 17 | 461 | 22 | 17 |
| 160 | 296 | 204 | 465 | 17 | 18 | 465 | 17 | 18 |

check whether an expression is deterministic, and differ only in the processing of stared subexpressions. When a starred subexpression has an error, each of the algorithms will detect one error.

Figure 2 shows the average time for detecting one nondeterministic regular expressions by the algorithms. Each value is obtained by the time to check the total amount of nondeterministic regular expressions in each size divided by the number of nondeterministic regular expressions in that size. The time used to print diagnostic information in the programs is not included. The algorithms spend averagely less than 6 milliseconds for a regular expression of size 160, thus are efficient in practice. It is not strange that the semantic checking algorithm runs faster, since the algorithms presented in this paper will do more than the semantic algorithm. On the other hand, the implementation of the algorithms presented in the paper still have much room for improvement. In the testing *deterministicpl* runs almost as faster as *deterministic_c* and *deterministic*. Thus we can use *deterministicpl* for diagnostic tasks.

## 5 Conclusion

Due to its semantic definition, a deterministic regular expression is hard to design and understand, and semantic checking techniques can only answer yes or no. The paper presented several algorithms as an attempt to diagnose nondeterministic regular expressions, making it possible to analyze and give hints to errors thus reducing the difficulty of designing deterministic content models. This would be convenient for designers to utilize their knowledge and intuition.

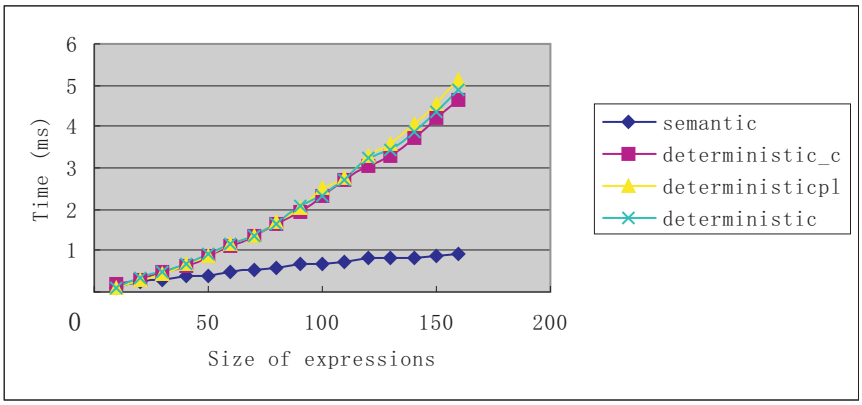


**Fig. 2.** Average running time of the algorithms

In the future, it would be useful to find more syntactic conditions for deterministic regular expressions, in the hope of more detailed revealing of errors. A presently unclear question is, can we use other more intuitive conditions to replace the emptiness condition of $followlast\text{-}first$ or $first\text{-}first$ intersection? The approaches to display diagnostic information effectively also constitute a significant aspect. The diagnostic information offered by the algorithms may also be used to generate counter examples of nondeterministic content models, which is also helpful for designers, but is not discussed in the paper. It is possible to integrate the above techniques in tools to provide analyzing and debugging facilities for content models.

## References

1. H. Ahonen. Disambiguation of SGML content models. PODP 1996, 27 – 37, 1996.
2. G. J. Bex, W. Gelade, W. Martens, F. Neven, Simplifying XML schema: effortless handling of nondeterministic regular expressions, SIGMOD 2009, 731 – 744, 2009.
3. G. J. Bex, F. Neven, and J. V. Bussche, DTDs versus XML schema: a practical study, WebDB 2004, 79 – 84, 2004.
4. A. Brüggemann-Klein, Regular expressions into finite automata, Theoretical Computer Science 120 (1993) 197 – 213.
5. A. Brüggemann-Klein and D. Wood, One-unambiguous regular languages, Information and Computation, 142(2):182 – 206, 1998.
6. V. M. Glushkov, The abstract theory of automata, Russian Math. Surveys 16 (1961) 1 – 53.
7. W. Martens, F. Neven, T. Schwentick, and G.J. Bex, Expressiveness and complexity of XML Schema, ACM Transactions on Database Systems, 31(3):770 – 813, 2006.
8. R. McNaughton, H. Yamada, Regular expressions and state graphs for automata, IRE Trans. on Electronic Computers 9 (1) (1960) 39 – 47.

## Proof of Proposition 2

The following lemma follows from the proof of Lemma 3.2 in [5].

**Lemma 4.** *Let $E = E_1^*$, $E_1$ be deterministic. Let $uxv, uyw \in L(\overline{E}), x, y \in \Sigma, u, v, w \in \Sigma^*$, such that $\overline{x} = \overline{y}$. If whenever $u = u_0 z, z \in last(\overline{E_1})$ and $x(y) \in first(\overline{E_1})$ we have $y(x) \in first(\overline{E_1})$, then $E$ is deterministic.*

Then we prove Proposition 2 as follows.

*Proof.* ($\Leftarrow$): Let $uxv, uyw \in L(\overline{E}), x, y \in \Sigma_{\overline{E}}, u, v, w \in \Sigma_{\overline{E}}^*, \overline{x} = \overline{y}$. If $u = u_0 z, z \in last(\overline{E_1})$ and $x \in first(\overline{E_1})$, then if $y \in followlast(\overline{E_1})$, then $x = y$, therefore $E$ is deterministic. Otherwise, $y \in first(\overline{E_1})$. Similarly, If $u = u_0 z, z \in last(\overline{E_1})$ and $y \in first(\overline{E_1})$, then if $x \in followlast(\overline{E_1})$, then $E$ is deterministic. Otherwise, $y \in first(\overline{E_1})$. Then, except the above two cases in which $E$ is deterministic, for other cases by Lemma 4 $E$ is deterministic.

($\Rightarrow$): Suppose $E_1$ is nondeterministic, then from Proposition 1 $E$ is nondeterministic. Suppose $\exists y_1 \in followlast(\overline{E_1}), \exists y_2 \in first(\overline{E_1}), \overline{y_1} = \overline{y_2}, y_1 \neq y_2$. We show that $E$ is nondeterministic. There are $y_2 u, vx, vxy_1 w \in L(\overline{E_1}), x \in \Sigma_{\overline{E_1}}, u, v, w \in \Sigma_{\overline{E_1}}^*$. Therefore, $vxy_2 u, vxy_1 w \in L(\overline{E})$, so $E$ is nondeterministic. □

## Proof of Proposition 3

*Proof.* (1) $\Rightarrow$ (2): We prove it by induction on the structure of $E$. The cases for $E = \varepsilon, \emptyset$ or $a, a \in \Sigma$ are obvious.

If $E = E_1 + E_2$, then $followlast(E) = followlast(E_1) \cup followlast(E_2)$ and $first(E) = first(E_1) \cup first(E_2)$. Let $y_1 \in followlast(\overline{E}), y_2 \in first(\overline{E})$. If $y_1 \in followlast(\overline{E_1}), y_2 \in first(\overline{E_1})$, then by induction $\mathcal{P}(E_1) = true$. Similarly, if $y_1 \in followlast(\overline{E_2}), y_2 \in first(\overline{E_2})$, then $\mathcal{P}(E_2) = true$. If $y_1 \in followlast(\overline{E_1}), y_2 \in first(\overline{E_2})$, then $y_1 \neq y_2$, thus by condition (1) $\overline{y_1} \neq \overline{y_2}$, i.e., $followlast(E_1) \cap first(E_2) = \emptyset$. Similarly, if $y_1 \in followlast(\overline{E_2}), y_2 \in first(\overline{E_1})$, we have $followlast(E_2) \cap first(E_2) = \emptyset$.

If $E = E_1 E_2$, then $followlast(E) = followlast(E_2)$ if $\neg(EPT(E_2))$ and $followlast(E_1) \cup followlast(E_2)$ otherwise, $first(E) = first(E_1)$ if $\neg(EPT(E_1))$ and $first(E_1) \cup first(E_2)$ otherwise. Let $y_1 \in followlast(\overline{E}), y_2 \in first(\overline{E})$. If $\neg(EPT(E_1)) \wedge \neg(EPT(E_2))$, then $y_1 \in followlast(\overline{E_2}), y_2 \in first(\overline{E_1})$, similarly as the above we have $followlast(E_2) \cap first(E_1) = \emptyset$. If $EPT(E_1) \wedge \neg(EPT(E_2))$, then $y_1 \in followlast(\overline{E_2}), y_2 \in first(\overline{E_1}) \cup first(\overline{E_2})$. If $y_2 \in first(\overline{E_1})$, then similarly $followlast(E_2) \cap first(E_1) = \emptyset$. If $y_2 \in first(\overline{E_2})$, then by induction $\mathcal{P}(E_2) = true$. Similarly we can prove the remaining cases.

If $E = E_1^*$, then $first(E) = first(E_1)$, $followlast(E) = followlast(E_1) \cup first(E_1)$. $\forall y_1 \in followlast(\overline{E_1}), \forall y_2 \in first(\overline{E_1})$, $y_1 \in followlast(\overline{E}), y_2 \in first(\overline{E})$, so by condition (1) if $\overline{y_1} = \overline{y_2}$ then $y_1 = y_2$. Then $\mathcal{P}(E_1) = true$.

(2) $\Rightarrow$ (1): We prove it by induction on the structure of $E$. The cases for $E = \varepsilon, \emptyset$ or $a, a \in \Sigma$ are obvious.

If $E = E_1 + E_2$, let $y_1 \in followlast(\overline{E}), y_2 \in first(\overline{E})$. If $y_1 \in followlast(\overline{E_1})$, $y_2 \in first(\overline{E_1})$, then since $\mathcal{P}(E_1) = true$, by induction if $\overline{y_1} = \overline{y_2}$ then $y_1 = y_2$. Similarly, if $y_1 \in followlast(\overline{E_2}), y_2 \in first(\overline{E_2})$, then if $\overline{y_1} = \overline{y_2}$ then $y_1 = y_2$. If $y_1 \in followlast(\overline{E_1}), y_2 \in first(\overline{E_2})$, then $y_1 \neq y_2$. On the other hand, since $followlast(E_1) \cap first(E_2) = \emptyset$, $\overline{y_1} \neq \overline{y_2}$, so condition (1) holds. Similarly we can prove the case when $y_1 \in followlast(\overline{E_2}), y_2 \in first(\overline{E_1})$.

If $E = E_1 E_2$, let $y_1 \in followlast(\overline{E}), y_2 \in first(\overline{E})$. If $\neg(EPT(E_1)) \wedge \neg(EPT(E_2))$, then $y_1 \in followlast(\overline{E_2}), y_2 \in first(\overline{E_1})$, then $y_1 \neq y_2$. Since $followlast(E_1) \cap first(E_2) = \emptyset$, $\overline{y_1} \neq \overline{y_2}$, so condition (1) holds. The other cases can be proved similarly.

If $E = E_1^*$, let $y_1 \in followlast(\overline{E}), y_2 \in first(\overline{E})$. If $y_1 \in followlast(\overline{E_1}), y_2 \in first(\overline{E_1})$, since $\mathcal{P}(E_1) = true$, by induction if $\overline{y_1} = \overline{y_2}$ then $y_1 = y_2$. If $y_1 \in first(\overline{E_1}), y_2 \in first(\overline{E_1})$, since $E_1$ is deterministic, if $\overline{y_1} = \overline{y_2}$ then $y_1 = y_2$. □