Checking Determinism of Regular Expressions with Counting [⋆]

Haiming Chen and Ping Lu

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

Abstract. We give characterizations of strong determinism for regular expressions with counting, based on which we present an $O(|\Sigma_E||E|)$ time algorithm to check whether an expression E with counting is strongly deterministic where Σ_E is the set of distinct symbols in E. It improves the previous upper bound of $O(|E|^3)$ time on the same decision problems for both standard regular expressions and regular expressions with counting. As a natural result of our work we derive a characterization of weak determinism for regular expressions with counting, which leads to a new $O(|\Sigma_E||E|)$ time algorithm for deciding weak determinism of regular expressions with counting.

1 Introduction

Regular expressions have been widely used in many applications. Different applications may require regular expressions with various extensions or restrictions, among them are deterministic regular expressions. For example, Document Type Definition (DTD) and XML Schema, which are the XML schema languages recommended by W3C, require that the content models should be *weakly deterministic regular expressions*. As another example, *strongly deterministic regular expressions* are used in query languages for XML streams [11]. Informally, weak determinism means that, when matching a word against an expression, a symbol can be matched to only one position in the expression without looking ahead. Meanwhile, strong determinism makes additional restriction that the use of operators should also be unique in the matching with the word. Weakly deterministic regular expressions have been studied in the literature, also under the name of *one-unambiguous* regular expressions [1, 3, 2, 5, 14, 13, 6, 12]. On the other hand, strong determinism (or strong one-unambiguity) of regular expressions has also attracted attentions recently [5, 14, 11, 7].

One basic problem is deciding weak or strong determinism of regular expressions. While deciding weak determinism of a standard regular expression E can be solved in $O(|\Sigma_E||E|)$ time [1] where Σ_E is the set of distinct symbols in

[⋆] Work supported by the National Natural Science Foundation of China under Grant No. 61070038.

E, deciding strong determinism of standard regular expressions is more involved and the up to date algorithm runs in $O(|E|^3)$ time [11]. Furthermore, it is known that deciding weak or strong determinism is nontrivial for regular expressions with *counting* (RE(#)) [5][1]. The latter is extended from standard regular expressions with iterative expressions (i. e., expressions of the form $E^{[m,n]}$), and is used for instance in XML Schema. For deciding weak determinism of regular expressions in RE(#) an $O(|\Sigma_E||E|)$ time method was given [8]. For deciding strong determinism of regular expressions in RE(#) an $O(|E|^3)$ time algorithm was presented [5]. In this paper we study properties of RE(#) and present characterizations of strong determinism for RE(#), based on which we give an $O(|\Sigma_E||E|)$ time algorithm to check whether an expression in RE(#) is strongly deterministic. Moreover our result can easily adapt to deciding weak determinism for RE(#) and trivially apply to deciding strong determinism for standard regular expressions, thus both gives a new $O(|\Sigma_E||E|)$ time algorithm for the former and improves the complexity bound from $O(|E|^3)$ time into $O(|\Sigma_E||E|)$ time for the latter.

Contributions. We give a structural characterization of strong determinism for RE(#). This characterization can lead to an $O(|\Sigma_E||E|)$ time algorithm. We give a further characterization of strong determinism for iterative expressions to achieve additional benefits. The new characterization elaborately distributes specific conditions for strong determinism of an iterative expression to some particular subexpressions of the expression. The benefits of the new characterization are that, it not only allows checking strong determinism in $O(|\Sigma_E||E|)$ time, but also enables deciding strong determinism of an iterative expression by particular subexpressions of the expression. Thus it is possible for instance that nondeterminism of the expression can be located locally and more precisely in a lower level subexpression.

Then we present an algorithm to check strong determinism for regular expressions in RE(#). The algorithm tests strong determinism directly on the original regular expressions and runs in time $O(|\Sigma_E||E|)$. As a natural result of our work we derive a characterization of weak determinism for RE(#), which gives rise to a new $O(|\Sigma_E||E|)$ time algorithm possessing similar features as above.

Related work. A majority of work considered determinism of standard regular expressions. Brüggemann-Klein [1] presented an algorithm for standard regular expressions to check if an expression is weakly deterministic based on Glushkov automata. By converting expressions into star normal form, the algorithm can check determinism in $O(|\Sigma_E||E|)$ time. In [4] a preliminary diagnosing algorithm was proposed for weak determinism of standard regular expressions which is based on testing expressions and runs in $O(|E|^2)$ time. The present work is inspired by that work, but here we deal with the different and more challenging problem of checking strong determinism for RE(#) and our algorithm takes $O(|\Sigma_E||E|)$ time. On the other hand, by applying techniques in this paper it is easy to improve the complexity of the algorithm in [4] into $O(|\Sigma_E||E|)$

[1] The nontrivialness is illustrated by an example in [5]: $(b?a^{[2,3]})^{[2,2]}b$ is weakly deterministic, but $(b?a^{[2,3]})^{[3,3]}b$ is not.

time. In [11] an $O(|E|^3)$ time algorithm was given to check strong determinism of standard regular expressions.

For expressions in RE(#), extensions of the Glushkov construction have been studied [5, 14, 9]. Relation between strong deterministic expressions and the corresponding Glushkov automata was set up [5], and a strong determinism checking algorithm was given, which runs in $O(|E|^3)$ time. Kilpeläinen [8] presented an $O(|\Sigma_E||E|)$ time algorithm to check weak determinism for RE(#).

The rest of the paper is organized as follows. Section 2 introduces definitions. In Section 3 the computation of some sets is discussed, which is prerequisite for the following algorithms. In Section 4 properties of regular expressions in RE(#) are studied. In Section 5 the characterizations of strong determinism for regular expressions in RE(#) are given, and an algorithm to check strong determinism of regular expressions in RE(#) is presented. In Section 6 the characterization of weak determinism for regular expressions in RE(#) derived from our work is presented. In Section 7 we show the local nondeterminism-locating feature of our characterizations by an example.

2 Preliminaries

Let Σ be an alphabet of symbols. The set of all finite words over Σ is denoted by Σ^*. The empty word is denoted by ε. The class of (standard) regular expressions over Σ, denoted by RE, is defined in the standard way: $\emptyset, \varepsilon$ or $a \in \Sigma$ is a regular expression, the union E_1+E_2, the concatenation E_1E_2, or the star E_1^* is a regular expression for regular expressions E_1 and E_2. Let $\mathbb{N}$ denote the set $\{0, 1, 2, \ldots\}$. The class of regular expressions with *counting*, denoted by RE(#), is extended from RE by further using the *numerical iteration operator*: $E^{[m,n]}$ is a regular expression for a regular expression E. The bounds m and n satisfy the following conditions: $m \in \mathbb{N}$, $n \in \mathbb{N}\backslash\{0\} \cup \{\infty\}$, and $m \leq n$. Notice $E^* = E^{[0,\infty]}$. Thus we do not need to separately consider the star operator in RE(#). Notice $E?$ is also used in content models, which is just an abbreviation of $E + \varepsilon$, and is therefore not separately considered in the paper.

For a regular expression E, the language specified by E is denoted by $L(E)$. The language of $E^{[m,n]}$ is defined as $L(E^{[m,n]}) = \bigcup_{i=m}^{n} L(E)^i$. Define $\lambda(E) = true$ if $\varepsilon \in L(E)$ and $false$ otherwise. An expression E is *nullable* if $\lambda(E) = true$. The size of a regular expression E in RE(#), denoted by $|E|$, is the number of symbols and operators occurring in E plus the sizes of the binary representations of the integers [5]. The symbols that occur in E, which form the smallest alphabet of E, will be denoted by Σ_E. An expression is in *normal form* if for its every nullable subexpressions $E_1^{[m,n]}$ we have $m = 0$ [5]. Expressions can be transformed into normal form in linear time [5]. Therefore, following [5], we assume expressions are in normal form in this paper.

For a regular expression we can mark symbols with subscripts so that in the marked expression each marked symbol occurs only once. For example $(a_1 + b_2)^{[6,7]}a_3b_4(a_5 + b_6)$ is a marking of the expression $(a + b)^{[6,7]}ab(a + b)$. The marking of E is denoted by $\overline{E}$. The same notation will also be used for dropping subscripts from the marked symbols: $\overline{\overline{E}} = E$. We extend the notation for words and sets of symbols in the obvious way. It will be clear from the context whether $\bar{\cdot}$ adds or drops subscripts.

Definition 1 ([3]). *An expression E is* weakly deterministic *if and only if, for all words $uxv, uyw \in L(\overline{E})$ where $|x| = |y| = 1$, if $x \neq y$ then $\overline{x} \neq \overline{y}$.*

The expression $a^{[0,2]}a$ is not weakly deterministic, since $a_2, a_1a_2 \in L(a_1^{[0,2]}a_2)$.

It is known that weakly deterministic regular expressions denote a proper subclass of regular languages [3].

A *bracketing of a regular expression E* is a labeling of the iteration nodes of the syntax tree by distinct indices [5]. The bracketing $\widetilde{E}$ of E is obtained by replacing each subexpression $E_1^{[m,n]}$ of E with a unique index i with $([_iE_1]_i)^{[m,n]}$. Therefore, a bracketed regular expression is a regular expression over alphabet $\Sigma \cup \Gamma_E$, where $\Gamma_E = \{[_i,]_i \mid 1 \leq i \leq |E|_\Sigma\}$, $|E|_\Sigma$ is the number of symbol occurrences in E. A string w in $\Sigma \cup \Gamma_E$ is correctly bracketed if w has no substring of the form $[_i]_i$.

Definition 2 ([5]). *A regular expression E is* strongly deterministic *if E is weakly deterministic and there do not exist strings u, v, w over $\Sigma \cup \Gamma_E$, strings $\alpha \neq \beta$ over Γ_E, and a symbol $a \in \Sigma$ such that $u\alpha av$ and $u\beta aw$ are both correctly bracketed and in $L(\widetilde{E})$.*

The expression $(a^{[1,2]})^{[1,2]}$ is weakly deterministic but not strongly deterministic. Both $[_2[_1a]_1]_2[_2[_1a]_1]_2$ and $[_2[_1a]_1[_1a]_1]_2$ are in $L(([_2([_1a]_1)^{[1,2]}]_2)^{[1,2]})$.

For an expression E over Σ, we define the following sets:

$first(E) = \{a \mid aw \in L(E), a \in \Sigma, w \in \Sigma^*\}$,

$followlast(E) = \{b \mid vbw, v \in L(E), v \neq \varepsilon, b \in \Sigma, w \in \Sigma^*\}$.

We assume expressions are reduced by the following rules: $E + \emptyset = \emptyset + E = E, E\emptyset = \emptyset E = \emptyset$, and $E\varepsilon = \varepsilon E = E$. For a reduced expression, it either does not contain $\emptyset$ or is $\emptyset$. Since we are not interested in the trivial case of an expression of $\emptyset$, in the following we assume an expression is not $\emptyset$.

3 Computing *followlast* sets

To determine the conditions given in the next sections, we will need to calculate the *first* and *followlast* sets and the λ function. The inductive definition of the λ function on expressions in RE is standard and can be found in, e. g., [1], which can be trivially extended to expressions in RE(#).

For any regular expression E, it is easy to see that *first* can be computed as follows.

$$\begin{aligned}
&first(\varepsilon) = \emptyset, first(a) = \{a\},\ a \in \Sigma_E;\\
&first(G + H) = first(G) \cup first(H);\\
&first(GH) = \begin{cases} first(G) \cup first(H) & \text{if } \varepsilon \in L(G),\\ first(G) & \text{otherwise;}\end{cases}\\
&first(G^{[m,n]}) = first(G).
\end{aligned} \tag{1}$$

The calculation of $followlast$ is however more involved. The following notion has been given in [10].

Definition 3 ([10]). *An iterative subexpression* $F = \overline{G}^{[m,n]}$ *of* $\overline{E}$ *is* flexible *in* $\overline{E}$, *denoted* $flexible(G^{[m,n]})$, *if there is some word* $uws \in L(\overline{E})$ *with* $w \in L(F)^l \cap L(\overline{G})^k$ *for some* $l \in \mathbb{N}$ *and* $k < l \times n$. *We call such a word* w *a witness to the flexibility of* F *in* $\overline{E}$.

The flexibility of an iterative expression can be computed in linear time [8].

For a marked expression $\overline{E}$, it is known that the following holds [8].

$$followlast(\varepsilon) = followlast(a) = \emptyset,\ a \in \Sigma_{\overline{E}}\ ;$$
$$followlast(\overline{G} + \overline{H}) = followlast(\overline{G}) \cup followlast(\overline{H});$$
$$followlast(\overline{GH}) = \begin{cases} followlast(\overline{G}) \cup first(\overline{H}) \cup followlast(\overline{H}) & \text{if } \varepsilon \in L(\overline{H}), \\ followlast(\overline{H}) & \text{otherwise;} \end{cases}$$
$$followlast(\overline{G}^{[m,n]}) = \begin{cases} followlast(\overline{G}) \cup first(\overline{G}) & \text{if } flexible(G^{[m,n]}) = true, \\ followlast(\overline{G}) & \text{otherwise.} \end{cases}$$

However, the above formula is incorrect for general expressions. For example, let $E = a + ab$. By definition, we have $followlast(E) = \{b\}$, since $a, ab \in L(E)$. But $followlast(a) = \emptyset$ and $followlast(ab) = \emptyset$, which means $followlast(E) \neq followlast(a) \cup followlast(ab)$. The remaining of this section deals with this issue.

The following lemma shows the relation between the considered sets on general and marked expressions.

Lemma 1. *Let* E *be a regular expression.*
(1) $\overline{followlast(\overline{E})} \subseteq followlast(E)$.
(2) $first(E) = \overline{first(\overline{E})}$.
(3) E *is weakly deterministic* $\Rightarrow followlast(E) = \overline{followlast(\overline{E})}$.

Then from Lemma 1 we have

Corollary 1. *For a weakly deterministic expression* E, $followlast$ *can be computed as follows.*

$$\begin{aligned} &followlast(\varepsilon) = followlast(a) = \emptyset, a \in \Sigma_E; \\ &followlast(G + H) = followlast(G) \cup followlast(H); \\ &followlast(GH) = \begin{cases} followlast(G) \cup first(H) \cup followlast(H) & \text{if } \varepsilon \in L(H), \\ followlast(H) & \text{otherwise;} \end{cases} \\ &followlast(G^{[m,n]}) = \begin{cases} followlast(G) \cup first(G) & \text{if } flexible(G^{[m,n]}) = true, \\ followlast(G) & \text{otherwise.} \end{cases} \end{aligned} \tag{2}$$

This gives computation of $followlast$ for weakly deterministic expressions.

Fortunately we will see later that in the algorithms only when E is weakly or strongly deterministic is $followlast(E)$ needed. Thus Equation (2) works for our purpose.

4 Properties of expressions in RE(#)

In this section we will develop further necessary properties. Fix an arbitrary coding on the syntax tree of an expression E in some ordering, such that each node in the syntax tree has a unique index. The subexpression corresponding to a node with index n is denoted by $E|_n$. Inside the syntax tree of E, the replacement of the subtree of a subexpression $E|_n$ with the syntax tree of an expression G is denoted by $E[E|_n \leftarrow G]$, which yields a new expression. For a subexpression $E|_n$ of E, let $E^{\flat}_{E|_n} = E[E|_n \leftarrow \flat E|_n \flat]$, where $\flat \notin \Sigma_E$.

Definition 4. *Let $\flat \notin \Sigma_E$. For a subexpression $E|_n$ of E, we say $E|_n$ is* continuing*, if for any words $w_1, w_2 \in L(E|_n)$, there are $u, v \in (\Sigma_E \cup \{\flat\})^*$, such that $u\flat w_1 \flat\flat w_2 \flat v \in L(E^{\flat}_{E|_n})$.*

If a subexpression $E|_n$ is continuing, we denote $ct(E|_n, E) = true$, otherwise $ct(E|_n, E) = false$. When there is no confusion, $ct(E|_n, E)$ is also written as $ct(E|_n)$. For the expression $E = (t(x+\varepsilon))^{[0,\infty]}l$, $ct(t) = true$, since $\flat t \flat\flat t \flat l \in L(E^{\flat}_t)$. Similarly, we can get $ct(x) = false$, $ct(x+\varepsilon) = false$, $ct(t(x+\varepsilon)) = true$, $ct((t(x+\varepsilon))^{[0,\infty]}) = false$, $ct(l) = false$ and $ct((t(x+\varepsilon))^{[0,\infty]}l) = false$.

Intuitively if a subexpression F of E is continuing then F is inside an iterative subexpression of E. We use $\flat$ in the definition to exclude the cases like $E = a^{[m,\infty]}$ where $ct(E) = false$ but $\forall x, y \in L(E), xy \in L(E)$, and $E = (a+b)(b+a)$ where $ct(b+a) = false$ but $\forall x, y \in L(b+a), xy \in L(E)$. Formally we offer a characterization of continuing in Proposition 1.

Let $E_1 \preceq E$ denote that E_1 is a subexpression of E. By $E_1 \prec E$ we denote $E_1 \preceq E$ and $E_1 \neq E$.

If $F \preceq G$ for some $G^{[m,n]} \preceq E$ $(n > 1)$, and there is no $G_1^{[m_1,n_1]} \preceq E$ $(n_1 > 1)$ such that $F \preceq G_1 \prec G$, we call $G^{[m,n]}$ the *lowest upper nontrivial iterative expression (LUN)* of F. Let $E = a^{[0,1]}b^{[1,2]}$, then E does not have a LUN, a does not have a LUN, and b has a LUN, that is $b^{[1,2]}$. It is easy to see if a subexpression is inside any iterative expression $G^{[m,n]}$ $(n > 1)$, then it has a LUN. Obviously a subexpression may not have a LUN, and if it has, its LUN is unique.

Proposition 1. *A subexpression F of E is continuing iff there exists the LUN $G^{[m,n]} \preceq E$ $(n > 1)$ of F, such that $L(\flat F \flat) \subseteq L(G^{\flat}_F)$.*

It is obvious that the *continuing* property of F is locally decided in the LUN of F. If F does not have the LUN, then F cannot be continuing. A related concept is the *factor* of an expression [10]. A continuing subexpression is a proper factor of its LUN.

Indeed if F is continuing, then F can affect the determinism of its LUN, which will be clear later. That is the reason why we study the *continuing* property. Below we first consider the calculation of the *continuing* property for subexpressions of an expression.

Clearly an expression E itself is not continuing, since there is not a LUN of E. That is,

Proposition 2. *$ct(E) = false$ for any expression E.*

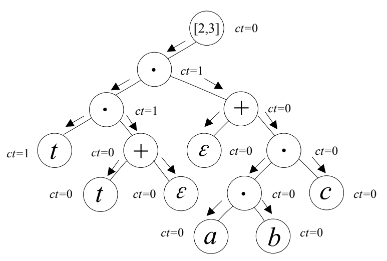

Fig. 1. The *continuing* property values in $((t(t+\varepsilon))(\varepsilon+(abc)))^{[2,3]}$ (0 for $false$ and 1 for $true$)

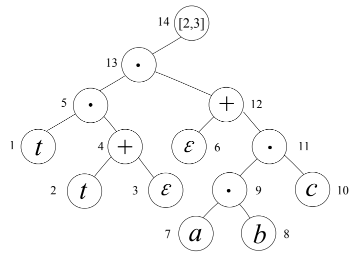

Fig. 2. Node processing sequences in $((t(t+\varepsilon))(\varepsilon+(abc)))^{[2,3]}$

We also have the following properties.

Proposition 3. *For a subexpression $F = H + I$ of E, $ct(H) = ct(I) = ct(F)$.*

Proposition 4. *For a subexpression $F = HI$ of E, $ct(H) = true$ (resp. $ct(I) = true$) iff $ct(F) = true$ and $\lambda(I) = true$ (resp. $\lambda(H) = true$).*

Proposition 5. *For a subexpression $F = G^{[m,n]}$ of E, if $n = 1$, then $ct(G) = ct(F)$, if $n > 1$, then $ct(G) = true$.*

Propositions 2–5 have given an algorithm to compute *continuing*. The *continuing* property of subexpressions of E is actually an attribute that is inherited from upper level subexpressions to lower level subexpressions. This means the *continuing* property can be computed in a top-down order on the structure of E.

From the above we can also have the following fact.

Fact. In the downward propagation of the *continuing* property on the nodes of the syntax tree of a regular expression E, only two kinds of nodes may change the value of *continuing* property, i. e., the node corresponding to a concatenation, called concat node, and the node corresponding to a iteration, called iteration node. Moreover, the concat nodes may change the *continuing* value from $true$ to $false$, while the iteration nodes may change the *continuing* value from $false$ to $true$.

The computation of the *continuing* value is showed by an example $E = ((t(t+\varepsilon))(\varepsilon+(abc)))^{[2,3]}$ from Figure 1.

5 Strong Determinism

A characterization of strong determinism is presented in the following.

Lemma 2. *Let E be a regular expression.*
(1) $E = \varepsilon$, or $a \in \Sigma$: E is strongly deterministic.

(2) $E = E_1 + E_2$: E is strongly deterministic iff E_1 and E_2 are strongly deterministic and $first(E_1) \cap first(E_2) = \emptyset$.
(3) $E = E_1E_2$: If $\varepsilon \in L(E_1)$, then E is strongly deterministic iff E_1 and E_2 are strongly deterministic, $first(E_1) \cap first(E_2) = \emptyset$, and $followlast(E_1) \cap first(E_2) = \emptyset$.
If $\varepsilon \notin L(E_1)$, then E is strongly deterministic iff E_1 and E_2 are strongly deterministic, and $followlast(E_1) \cap first(E_2) = \emptyset$.
(4) $E = E_1^{[m,n]}$: (a) If $n = 1$, E is strongly deterministic iff E_1 is strongly deterministic.
(b) If $n > 1$, E is strongly deterministic iff E_1 is strongly deterministic and $followlast(E_1) \cap first(E_1) = \emptyset$.

Lemma 2 can lead to an $O(|\Sigma_E||E|)$ time algorithm to check strong determinism by using similar techniques as introduced later. Furthermore, based on the *continuing* property, we give a new characterization of strong determinism of iterative expressions in the following, which not only allows checking strong determinism in $O(|\Sigma_E||E|)$ time, but also has additional benefits that will be presented in Section 7.

The boolean-valued function $\mathcal{S}$ on RE(#), introduced in the following, will be used to check strong determinism of expressions by using the *continuing* property.

Definition 5. *The boolean-valued function $\mathcal{S}(E)$ is defined as*

$$\begin{aligned}
&\mathcal{S}(\varepsilon) = \mathcal{S}(a) = true \quad a \in \Sigma \\
&\mathcal{S}(E_1 + E_2) = \mathcal{S}(E_1) \wedge \mathcal{S}(E_2) \wedge (ct(E_1 + E_2) = false \\
&\qquad\qquad \vee (followlast(E_1 + E_2) \cap first(E_1 + E_2) = \emptyset)) \\
&\mathcal{S}(E_1E_2) = \mathcal{S}(E_1) \wedge \mathcal{S}(E_2) \wedge (ct(E_1E_2) = false \\
&\qquad\qquad \vee (first(E_1E_2) \cap followlast(E_1E_2) \ = \emptyset)) \\
&\mathcal{S}(E_1^{[m,n]}) = \mathcal{S}(E_1) \wedge (ct(E_1^{[m,n]}) = false \\
&\qquad\qquad \vee (first(E_1^{[m,n]}) \cap followlast(E_1^{[m,n]}) \ = \emptyset))
\end{aligned}$$

In fact, and somewhat surprisingly, the function $\mathcal{S}$ gives exactly the specific conditions that E_1 should satisfy besides the condition of E_1 being strongly deterministic, to ensure an iterative expression $E_1^{[m,n]}$ to be strongly deterministic. This is shown in Propositions 6 and 7.

Proposition 6. *Let a subexpression E of an expression be strongly deterministic. We have $(first(E) \cap followlast(E) = \emptyset \vee ct(E) = false) \Leftrightarrow \mathcal{S}(E) = true$.*

Proposition 7. *For $E = E_1^{[m,n]}$ $(n > 1)$, E is strongly deterministic iff E_1 is strongly deterministic and $\mathcal{S}(E_1) = true$.*

The function $\mathcal{S}$ allows for one-pass computation on the syntax tree of the expression. This is a good property from at least the algorithmic point of view.

We can then derive an algorithm from the characterization given in Proposition 7 and Lemma 2, as follows.

Algorithm 1 *Is_Strong_Det*

```
Input: a regular expression in RE(#), $E$
Output: true if $E$ is strongly deterministic or false otherwise
 1. return $Strong\_DET(E, \text{false})$
```

Procedure 1 *Strong_DET*(E, *continuing*)

```
Input: a regular expression in RE(#), $E$, and a Boolean value $continuing = ct(E)$
Output: true if $E$ is strongly deterministic or false otherwise
 1. if $E = \varepsilon$ then
 2.    $first(E) \leftarrow \emptyset; followlast(E) \leftarrow \emptyset$; return true
 3. if $E = a$ for $a \in \Sigma$ then
 4.    $first(E) \leftarrow \{a\}; followlast(E) \leftarrow \emptyset$; return true
 5. if $E = E_1 + E_2$ then
 6.    $d_1 \leftarrow continuing$; $d_2 \leftarrow continuing$
 7.    if $Strong\_DET(E_1,d_1) \wedge Strong\_DET(E_2,d_2)$ then
 8.       if $first(E_1) \cap first(E_2) \neq \emptyset$ then
 9.          return false
10.       Calculate $first(E), followlast(E)$ (Equations (1), (2))
11.       if $continuing \wedge (first(E) \cap followlast(E) \neq \emptyset)$ then
12.          return false
13.       return true
14.    else return false
15. if $E = E_1E_2$ then
16.    Calculate $d_1, d_2$ by Proposition 4
17.    if $Strong\_DET(E_1,d_1) \wedge Strong\_DET(E_2,d_2)$ then
18.       if $followlast(E_1) \cap first(E_2) \neq \emptyset$ then
19.          return false
20.       if $\lambda(E_1) \wedge first(E_1) \cap first(E_2) \neq \emptyset$ then
21.          return false
22.       Calculate $first(E), followlast(E)$ (Equations (1), (2))
23.       if $continuing \wedge (first(E) \cap followlast(E) \neq \emptyset)$ then
24.          return false
25.       return true
26.    else return false
27. if $E = E_1^{[m,n]}$ then
28.    if $n = 1$ then $d_1 \leftarrow continuing$ else $d_1 \leftarrow$ true
29.    if $Strong\_DET(E_1,d_1)$ then
30.       Calculate $first(E), followlast(E)$ (Equations (1), (2))
31.       if $continuing \wedge (first(E) \cap followlast(E) \neq \emptyset)$ then
32.          return false
33.       return true
34.    else return false
```

First the syntax tree of a regular expression can be constructed and the λ function can be evaluated during the construction in linear time [1]. Then by carefully arranging the computation, all of the other computation can be done in one run on the syntax tree.

There are mainly the following kinds of work that should be completed by the algorithm: (1). Compute the *continuing* property. (2). Examine mixed test conditions. As mentioned before $\mathcal{S}$ represents specific conditions for subexpressions of iterative subexpressions. The algorithm should combine $\mathcal{S}$ with other conditions in Lemma 2. (3). Compute $first, followlast$ sets.

Work (1) can be done in a top-down manner on the syntax tree of the expression. Work (2) and (3) can be done at the same time in a bottom-up and incremental manner on the syntax tree. In this way, according to the conditions in Lemma 2, when current subexpression E_1 is tested to be nondeterministic then the expression is nondeterministic and the computation of $first$ and $followlast$ sets for E_1 is not necessary. Putting the above together, all the computation can be completed in one pass on the syntax tree, by a top-down then bottom-up traversal.

Actually, from the point of view of attribute grammars, the *continuing* property is precisely an inherited attribute, while the $first, followlast$ sets, and the determinism of subexpressions are all synthesized attributes. All the computation can be completed by attribute evaluation in one pass.

The algorithm *Is_Strong_Det*(E) takes as input a regular expression, and outputs a Boolean value indicating if the expression is strongly deterministic.

Theorem 1. *Is_Strong_Det(E) returns true iff E is strongly deterministic.*

There are at most $O(|E|)$ nodes in the syntax tree of E. In the algorithm, the calculation of $first$ and $followlast$ sets is done at the same time with determinism test in a bottom-up and incremental manner, and can be computed on the syntax tree of E in $O(2|\Sigma_E||E|)$ time. Emptiness test of $first(E_1) \cap first(E_2)$ or $followlast(E_1) \cap first(E_2)$ for subexpressions E_1, E_2 can be completed in $O(2|\Sigma_E|)$ time with an auxiliary array indexed by every symbols in the alphabet of E. The algorithm may conduct the test at every inner node on a bottom-up traversal of the syntax tree of E, which totally takes $O(2|\Sigma_E||E|)$ time. So the time complexity of the algorithm is $O(|\Sigma_E||E|)$. For a fixed alphabet, the algorithm has linear running time. Hence

Theorem 2. *Is_Strong_Det(E) runs in time* $O(|\Sigma_E||E|)$.

6 Adaption to weak determinism

First consider the following relatively easy fact which still relies on marked expressions.

Lemma 3 ([3, 8]). *Let E be a regular expression.*
(a) $E = \varepsilon$ or $a \in \Sigma$: then E is weakly deterministic.

(b) $E = E_1+E_2$: E is weakly deterministic iff E_1 and E_2 are weakly deterministic and $first(E_1) \cap first(E_2) = \emptyset$.
(c) $E = E_1E_2$: (1) If $\varepsilon \in L(E_1)$, then E is weakly deterministic iff E_1 and E_2 are weakly deterministic, $first(E_1) \cap first(E_2) = \emptyset$, and $followlast(E_1) \cap first(E_2) = \emptyset$.
(2) If $\varepsilon \notin L(E_1)$, then E is weakly deterministic iff E_1 and E_2 are weakly deterministic and $followlast(E_1) \cap first(E_2) = \emptyset$.
(d) $E = E_1^{[m,n]}$: (1) If $n = 1$, then E is weakly deterministic iff E_1 is weakly deterministic; (2) If $n > 1$, then E is weakly deterministic iff E_1 is weakly deterministic and $\forall x \in followlast(\overline{E_1})$, $\forall y \in first(\overline{E_1})$, if $\overline{x} = \overline{y}$ then $x = y$.

We can use the *continuing* property to improve the characterization of weak determinism of iterative expressions as before. Let $\varphi(E) = \forall x \forall y(x \in followlast(\overline{E}) \wedge y \in first(\overline{E}) \wedge \overline{x} = \overline{y} \rightarrow x = y)$.

Definition 6. *The boolean-valued function $\mathcal{W}(E)$ is defined as*

$$\begin{aligned}
&\mathcal{W}(\varepsilon) = \mathcal{W}(a) = true \quad a \in \Sigma\\
&\mathcal{W}(E_1 + E_2) = \mathcal{W}(E_1) \wedge \mathcal{W}(E_2) \wedge (ct(E_1 + E_2) = false \vee\\
&\qquad (followlast(E_1) \cap first(E_2) = \emptyset \wedge\\
&\qquad followlast(E_2) \cap first(E_1) = \emptyset))\\
&\mathcal{W}(E_1E_2) = \mathcal{W}(E_1) \wedge \mathcal{W}(E_2) \wedge (ct(E_1E_2) = false \vee\\
&\qquad (first(E_1) \cap followlast(E_2) = \emptyset \wedge\\
&\qquad (\lambda(E_1) \vee \neg\lambda(E_2) \vee first(E_1) \cap first(E_2) = \emptyset)))\\
&\mathcal{W}(E_1^{[m,n]}) = \mathcal{W}(E_1)
\end{aligned}$$

Proposition 8. *Let a subexpression E of an expression be weakly deterministic. We have $(\varphi(E) = true \vee ct(E) = false) \Leftrightarrow \mathcal{W}(E) = true$.*

Proposition 9. *For $E = E_1^{[m,n]}$ $(n > 1)$, E is weakly deterministic iff E_1 is weakly deterministic and $\mathcal{W}(E_1) = true$.*

From the above analysis and using similar techniques for *Is_Strong_Det*(E), we get an algorithm *DCITER* to check weak determinism of regular expressions, which runs in time $O(|\Sigma_E||E|)$. For the limited space the concrete algorithm is not presented in the paper.

7 The local nondeterminism-locating feature and discussion

Below we show the local nondeterminism-locating feature of our methods by an example. We use the method *DCITER* here to compare with the existing algorithm *linear UPA* [8] for deciding weak determinism. We use the same expression $E = ((t(t + \varepsilon))(\varepsilon + (abc)))^{[2,3]}$ as in the previous example. The syntax tree of E is showed in Figure 2.

In *linear UPA*, the sequence of the processed nodes is $1 \to 2 \to \ldots \to 14$. At node 14, the algorithm will find that E is not deterministic, and then terminate.

In *DCITER*, the sequence of the processed nodes is $1 \to 2 \to 3 \to 4 \to 5$. At node 5, because the *continuing* value at the node is *true* and $t \in first(t)$ and $t \in first(t + \varepsilon)$, the algorithm will report an error immediately and terminate.

The example clearly shows that an iterative expression E can be nondeterministic while all its subexpressions are deterministic, and, moreover, in this situation *DCITER* may find the nondeterminism locally by checking subexpressions of E, in this case $t(t + \varepsilon)$. On the contrary *linear UPA* can only find this nondeterminism after examining the whole expression. So we can see that our methods can locate errors more precisely. This suggests our methods are also more advantageous for diagnosing purpose.

Acknowledgments The authors thank the anonymous reviewers for their valuable comments that helped us to improve the presentation of the paper.

References

1. A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.
2. A. Brüggemann-Klein and D. Wood. Deterministic regular languages. In *STACS 92*, pages 173–184. Springer-Verlag, 1992.
3. A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.
4. H. Chen and P. Lu. Assisting the design of XML Schema: diagnosing nondeterministic content models. In *APWeb 2011*, volume 6612 of *Lecture Notes in Computer Science*, pages 301–312. Springer Berlin/Heidelberg, 2011.
5. W. Gelade, M. Gyssens, and W. Martens. Regular expressions with counting: weak versus strong determinism. *SIAM J. Comput.*, 41(1):160–190, 2012.
6. W. Gelade and F. Neven. Succinctness of the complement and intersection of regular expressions. *STACS 2008*, pages 325–336, 2008.
7. D. Hovland. The membership problem for regular expressions with unordered concatenation and numerical constraints. In *LATA 2012*, volume 7183 of *Lecture Notes in Computer Science*, pages 313–324. Springer Berlin/Heidelberg, 2012.
8. P. Kilpeläinen. Checking determinism of XML Schema content models in optimal time. *Informat. Systems*, 36(3):596–617, 2011.
9. P. Kilpeläinen and R. Tuhkanen. Towards efficient implementation of XML Schema content models. In *DocEng'04*, pages 239–241, New York, NY, USA, 2004. ACM.
10. P. Kilpeläinen and R. Tuhkanen. One-unambiguity of regular expressions with numeric occurrence indicators. *Information and Computation*, 205(6):890–916, 2007.
11. C. Koch and S. Scherzinger. Attribute grammars for scalable query processing on XML streams. *The VLDB Journal*, 16(3):317–342, 2007.
12. W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *MFCS 2004*, pages 889–900. Springer, 2004.
13. 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.
14. C. M. Sperberg-McQueen. Notes on finite state automata with counters. *http://www.w3.org/XML/2004/05/msm-cfa.html*, 2004.