ISCAS-SKLCS-18-02 February 2018

中 国 科 学 院 软 件 研 究 所
计 算 机 科 学 实 验 室 报 告

Towards an Effective Syntax for Deterministic Regular Expressions

by

Haiming Chen, Ping Lu, Zhiwu Xu

State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing 100190 China

Towards an Effective Syntax for Deterministic Regular Expressions

Haiming Chen
State Key Laboratory of Computer Science, Institute of Software
Chinese Academy of Sciences, Beijing 100190, China

Ping Lu
BDBC, Beihang University

Zhiwu Xu
College of Computer Science and Software Engineering
Shenzhen University, Shenzhen 518060, China

Abstract

Deterministic regular expressions (DREGs) are a core part of XML Schema and used in other applications. But unlike regular expressions, DREGs do not have a simple syntax, instead they are defined in a semantic manner, which puts a burden on the user to develop XML Schema Definitions and to use DREGs. In this paper, we propose a syntax for DREGs, and prove that the class of DREGs is context-free. Next, one of the burdens for using DREGs is the automatic generation of DREGs which is indispensable in practice. Based on the context-free grammars for DREGs, we further design a generator for DREGs, which can generate sentences randomly. Experimental results demonstrate the efficiency and usefulness of the generator.

1 Introduction

Deterministic regular expressions (DREGs) are a mystery to users. They are a core part of XML Schema [1, 2] and used in other applications (e.g., [3, 4]). However, unlike regular expressions, they do not have a simple syntax. Instead they are defined in a semantic manner, which means that to check whether a regular expression is deterministic, we need to verify that for every two words w_1 and w_2 in the language represented by the regular expression, whether w_1 and w_2 satisfy some condition (see Definition 1). This puts a burden on the user to develop XML Schema Definitions (XSDs) and use DREGs. DREGs have been studied in the literature, also under the name of one-unambiguous regular expressions, e.g., [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19].

Grammars for DREGs and their properties. In this paper, we propose a syntax for DREGs in terms of a set of parameters. Although regular expressions are context-free and DREGs are a proper subset of regular expressions, DREGs are not necessarily context-free. Since not all subsets of regular expressions are context-free. A well-known example is the set $\{a^n b^n c^n | n \in \mathcal{N}\}$, where $\mathcal{N}$ represents the set of all positive natural numbers. This set of strings forms a context-sensitive language, and is not context-free.

We show that the class of DREGs is context-free, $i.e.$, the class of DREGs can be characterized by context-free grammars. As far as we know, this is the first time to give a syntax for DREGs and prove the class of DREGs is context-free.

Lack of a syntax for DREGs has long troubled researchers, yet it remains unsolved today (although there has been a lot of work on constructing DREGs for users from different perspectives, e.g., [6, 20, 12, 9, 10, 21, 22, 23, 24], see related work for details.). The basis for it is how to give generating systems for DREGs that can be used to construct grammars. This basically requires that the characterization of DREGs should be given for the original expressions, rather than the marked expressions (see Section 2 for the definition), while the marked expressions are extensively studied in the literature [6, 25, 26]. In a series of work [9, 10], Chen and Lu had proposed a new method for checking determinism of DREGs, which is based on the original expressions. This becomes the basis of the syntax for DREGs presented in this paper.

The construction of the syntax is achieved by building inference systems for DREGs, which are then used for giving grammars for DREGs. In concrete, based on properties given in [9, 10], we first obtain inference systems for DREGs. Then, starting from the inference systems, we construct grammars for DREGs, where each nonterminal symbol has a set of parameters, and each production satisfy a set of equations of these parameters between the left and right nonterminals. In this way, we have actually defined a class of grammars, and the grammars with alphabets of the same size are isomorphic.

We prove that the grammars exactly defines DREGs, and are context-free. Furthermore, by using the pumping lemma for regular languages [27], we prove that the class of DREGs cannot be defined by regular grammars. Then we conclude that the class of DREGs is context-free.

To effectively use the grammar, we further find some necessary and sufficient conditions for valid productions (see next section for its definition) which can reduce the number of productions in the grammars. For example, by using these conditions the number of productions for the alphabet size of 3 drops from 43698 to 14904. This indicates that optimizations are quite useful for the context-free grammars for DREGs.

The automatic generation of DREGs. Next, we consider how to facilitate the use of DREGs with the help of the context-free grammars for DREGs. Here we consider the automatic generation of DREGs. Automatic generation of DREGs is indispensable in many applications such as the testing, experiments of programs having input of DREG types, data experiments having a large number of DREG sentences, etc. For example, suppose that the user has a program for checking inclusion of Document Type Definitions (DTDs) [1], which basically reduces to a program A for checking inclusion of DREGs: $\mathsf{Bool}\ A(b : \text{DREG},\ c : \text{DREG})$, where b and c denote the content models of two DTDs, which are required to be deterministic [1]. The program A takes two DREGs as input, and returns true if $b \subseteq c$ and false otherwise. And the user needs many different DREGs to test the program. Since DREGs do not have a simple syntax before, one possible way to do this is first to use a sentence generator for regular expressions, then for each generated regular expression, to decide whether it is deterministic. Since the ratio of DREGs in regular expressions is low, within a certain period of time the user may not be able to obtain enough DREG sentences as needed (e.g., the ratio of DREGs is less than 1% for regular expressions of length 50 with alphabet size 40 [9]). Furthermore, if the user also has requirements on the structure of DREGs, such as lengths of DREGs, the generation will be more difficult.

Based on the context-free grammars for DREGs, we design a generator for DREGs. A naive idea is to construct the grammar first and then use the existing methods [28, 29, 30, 31, 32] to randomly generate sentences from the constructed grammar. However, because the grammar is too large to construct fully (see Section 3.2 and Table 1) and existing methods require some pre-processings of the whole grammar, the existing methods become inefficient and thus are not suitable for the grammars of DREGs. In existing methods, the pre-processings are to keep the information needed for randomness, thus contribute to make the random generation efficient. Here the key problem is how to make the random generation efficient without the construction and the pre-processings of the whole grammar. Thanks to the information in symbols in the grammars of DREGs, *i.e.*, the parameters of the nonterminal symbols, we can take full advantage of this information to make the random generation efficient without the construction and the pre-processings of the whole grammar. Our solution consists of the following ideas: (1) construct the valid productions only when they are needed, to avoid the construction of the whole grammar; (2) impose a conservative length condition to make the generation efficient; (3) employ a length control mechanism to improve the efficiency of the generation.

Experiments of the generator. Finally, we performed several experiments to evaluate our generator. The results demonstrate the efficiency and usefulness of our random generator.

Contributions. The contributions of the paper are listed as follows.

(1) We propose the first syntax for DREGs (Section 3), and prove that the class of DREGs is context-free. Meanwhile, by using the pumping lemma for regular languages [27], we also prove that the class of DREGs cannot be defined by regular grammars.

(2) We give necessary and sufficient conditions for valid productions (Section 4). The experiments show that these conditions are quite useful for reducing the size of the context-free grammars for DREGs.

(3) Based on the grammars, we further design and implement a random genera-

tion algorithm for DREGs (Section 5), in which productions are only constructed when they are needed. It further imposes a conservative length condition and a length control mechanism to ensure efficiency.

(4) We experimentally evaluate the generator (Section 6). The results show although the grammars can be exponentially large, by constructing the grammar in an on-the-fly manner, our random generator can generate DREGs efficiently. Moreover, we demonstrate that the generators for regular expressions are not feasible for generating DREGs.

Related work. The related work is categorized as follows.

Deterministic regular expressions. To decide whether a standard expression is deterministic, Brüggemann-Klein [5] gave an $O(|\Sigma_E||E|)$ time algorithm, which is based on the Glushkov automaton of the expression. For expressions with counting, Kilpeläinen [25] presented an $O(|\Sigma_E||E|)$ time algorithm by examining the marked expression. Based on [9] and [25], Chen and Lu also gave an $O(|\Sigma_E||E|)$ time algorithm for checking determinism of expressions with counting [10]. The main difference between their work and the work in [25] is that the algorithm in [10] is based on the original expressions, while the algorithm in [25] is based on marked expressions. Groz and Maneth [13] gave the first $O(|E|)$ time algorithm for checking determinism of standard regular expressions and expressions with counting. Peng *et al.* [19] gave an $O(|\Sigma_E||E|)$ time algorithm for checking determinism of expressions with interleaving.

There has been a lot of work on constructing DREGs for users, e.g., [6, 20, 12, 9, 10]. In [6], Brüggemann-Klein and Wood provided an exponential time algorithm to decide whether a regular language, given by an arbitrary regular expression, is deterministic. An algorithm was further given there to construct equivalent DREGs for nondeterministic expressions when possible, while the resulting expression could be double-exponentially large. Bex *et al.* [12] had optimized this construction algorithm. For nondeterministic regular languages, Ahonen [20] proposed an algorithm to construct approximate DREGs. Bex *et al.* [12] had also developed algorithms to approximate regular expressions. But here the languages is approximated, and solutions are needed to have the same language. Chen and Lu [9] had preliminarily explored the diagnosing problem of checking determinism of standard regular expressions. In [10] they improved the method for regular expressions with counting.

Another way to help users is to infer DREGs from sample strings. Bex *et al.* [21] gave algorithms to learn SOREs and CHAREs from example words. Here SOREs stand for single occurrence regular expressions, and CHAREs stand for chain regular expressions. Both of these expressions are subclasses of DREGs. Later, they gave methods to infer XML Schema from XML documents [22]. In a later paper [23], they provided algorithms to learn deterministic k-occurrence regular expressions, which are based on the Hidden Markov Model. Recently, Freydenberger and Kötzing [24] gave a linear time algorithm to infer SOREs and CHAREs from samples. Moreover, their algorithm generates the optimal SOREs and CHAREs [24].

The definability problem of DREGs is to decide, whether a given regular expression can be expressed by an equivalent DREG, which was proved to be PSPACE-complete [12, 18, 16]. Latte and Niewerth [17] proved whether a standard regular expression is equivalent to any deterministic regular expression with counting is decidable in 2EXPSPACE, whereas the exact complexity is still open. Lu *et al.* [14] showed whether a given standard regular expression on unary alphabet can be expressed by a DREG is coNP-complete. In [11] the descriptional complexity of DREGs has been studied. Gelade *et al.* [33] examined the descriptional complexity of intersection for DREGs and proved that the exponential bound on intersection is tight for DREGs.

Sentence Generation. Hanford [34] presented the first algorithm for generating sentences randomly from context-free grammars. Arnold and Sleep [28] considered the uniformity of random sentences of length n, where uniformity means that all strings of length n are generated by the grammar equally, and presented a linear algorithm to generate balanced parenthesis strings uniformly. Hickey and Cohen [35] presented two algorithms to generate sentences of length n uniformly at random from a general context-free grammar. Followed up, researchers proposed several uniform random generation algorithms to improve either the time and space bounds [29, 36] or the pre-processing [30]. According to the frequencies of letters, Denise *et al.* [31] proposed two other uniform random generation algorithms. Some researchers considered other types of grammars, such as Bertoni *et al.* [37] took the ambiguity into account, while Ponty and his collaborators [38, 39] considered the weighted context-free grammars. Besides, Héam and Masson [40] presented a uniform random generation algorithm using pushdown automata. Dong's enumeration algorithm [32] can be used as another random generation algorithm as well [41]. However, to make the generation efficient and/or to ensure the uniformity, all these algorithms require some pre-processings on the whole grammar, which are inefficient and hard for our grammars. By taking full advantage of the information that the symbols in the DREG grammars take, our algorithm needs no pre-processings. Moreover, our random generation focus the uniformity on productions of a same nonterminal symbol and to generate sentence of length no longer than n, so it may lose the uniformity on sentences of length n. As a future work, we will consider the distribution on sentences of length n by taking the symmetric similarity of DREG grammars into account.

Besides random generation, some studies adopted other strategies, like coverage criteria [42, 43], length control [44], and sentence enumeration [32, 45].

For random generation of XSD, Antonopoulos *et al.* [46] initiated the work of uniform XSD generation by developing an algorithm for random generation of k-occurrence automata (k-OAs) for contend models. However they did not manage to do random sampling of DREGs. Since content models of XSDs are DREGs, and languages accept by k-OAs do not equal to the ones for DREGs, our grammars for DREGs and automatic generation of DREGs fills a gap in and is a nice addition to [46].

2 Definitions

Let Σ be an alphabet of symbols. The set of all finite words over Σ is denoted by Σ^*. A regular expression over Σ is ε, $a \in \Sigma$, the union $r_1 \mid r_2$, the concatenation $r_1 \cdot r_2$, the plus r_1^+, or the question mark $r_1?$ for regular expressions r_1 and r_2. Here ε represents the empty word. Note that r^* (the Kleene star) is an abbreviation of $\varepsilon \mid r^+$, and we will not consider regular expressions like r^* in the following. For a regular expression r, the language specified by r is denoted by $L(r)$. Let $\lambda(r) = \mathsf{true}$ if $\varepsilon \in L(r)$, or false otherwise. The inductive computation of λ was given in [5]. The symbol occurrences in r is denoted as $occur(r)$. The size of r is the number of symbol occurrences in r or the alphabetic width of r, denoted by $|r|$.

Next we define deterministic regular expressions (DREGs). To this end, we need some notations. For a regular expression we can mark symbols with subscripts so that in the marked expression each marked symbol occurs only once. For example, $(a_1 \mid b_2)^+ a_3 b_4 (a_5 \mid b_6)$ is a marking of the expression $(a \mid b)^+ ab(a \mid b)$. The marking of an expression r is denoted by r'. Accordingly the result of dropping the subscripts from a marked expression r is denoted by $r^\natural$. And then we have $(r')^\natural = r$. We extend the notation for words in an obvious way. By using these notations, we can define DREGs as follows.

Definition 1 ([6])**.** *An expression r is deterministic if it satisfies the following condition: for any two words $uxv, uyw \in L(r')$ with $|x| = |y| = 1$, if $x \neq y$, then $x^\natural \neq y^\natural$ holds. A regular language is deterministic if it is denoted by some deterministic expression.*

For example, the expression $(a \mid \varepsilon)((c \mid d)^+ \mid \varepsilon)(a \mid \varepsilon)$ is not deterministic, since $a_1, a_4 \in L((a_1 \mid \varepsilon)((c_2 \mid d_3)^+ \mid \varepsilon)(a_4 \mid \varepsilon))$, $a_1 \neq a_4$, but $(a_1)^\natural = (a_4)^\natural$. For this example, all words u, v, and w are ε. It is known that the class of DREGs denotes a proper subclass of regular languages [6].

Then we introduce some necessary components that will be used in our grammars. For an expression r over Σ, we define the following sets:

$$\mathsf{First}(r) = \{a \mid aw \in L(r), a \in \Sigma, w \in \Sigma^*\}$$
$$\mathsf{followLast}(r) = \{b \mid vbw, v \in L(r), v \neq \varepsilon, b \in \Sigma, w \in \Sigma^*\}$$

The computations for First and $\mathsf{followLast}$ can be founded in [5, 25, 10]. Intuitively, $\mathsf{First}(r)$ contains all the first symbols of words in $L(r)$, while $\mathsf{followLast}(r)$ contains all symbols b which can be appended to a word v in $L(r)$ to form another word vbw in $L(r)$. For the completeness of the paper, all rules for the computations of λ, First, and $\mathsf{followLast}$ are listed as follows. The computation of λ [5]:

$$\begin{aligned}
&\lambda(\varepsilon) = \mathsf{true};\\
&\lambda(a) = \mathsf{false},\ a \in \Sigma;\\
&\lambda(r \mid s) = \lambda(r) \vee \lambda(s);\\
&\lambda(r \cdot s) = \lambda(r) \wedge \lambda(s);\\
&\lambda(r?) = \lambda(r^*) = \mathsf{true};\\
&\lambda(r^+) = \lambda(r).
\end{aligned}$$

The computation of First [5]:

$$\begin{aligned}
&\mathsf{First}(\varepsilon) = \emptyset, \mathsf{First}(a) = \{a\},\ a \in \Sigma;\\
&\mathsf{First}(r \mid s) = \mathsf{First}(r) \cup \mathsf{First}(s);\\
&\mathsf{First}(r \cdot s) = \begin{cases} \mathsf{First}(r) \cup \mathsf{First}(s) & \text{if } \lambda(r) = \mathsf{true},\\ \mathsf{First}(r) & \text{otherwise;} \end{cases}\\
&\mathsf{First}(r^+) = \mathsf{First}(r)\\
&\mathsf{First}(r?) = \mathsf{First}(r).
\end{aligned}$$

The computation of $\mathsf{followLast}$ [25, 10]:

$$\begin{aligned}
&\mathsf{followLast}(\varepsilon) = \mathsf{followLast}(a) = \emptyset,\ a \in \Sigma;\\
&\mathsf{followLast}(r \mid s) = \mathsf{followLast}(r) \cup \mathsf{followLast}(s);\\
&\mathsf{followLast}(r \cdot s) = \begin{cases} \mathsf{followLast}(r) \cup \mathsf{First}(s) & \\ \quad \cup\, \mathsf{followLast}(s) & \text{if} \lambda(s) = \mathsf{true},\\ \mathsf{followLast}(s) & \text{otherwise;} \end{cases}\\
&\mathsf{followLast}(r^+) = \mathsf{followLast}(r) \cup \mathsf{First}(r);\\
&\mathsf{followLast}(r?) = \mathsf{followLast}(r).
\end{aligned}$$

Note that only for marked regular expressions and DREGs, can the $\mathsf{followLast}$ sets be computed in this way. For general regular expressions, it is not correct to compute $\mathsf{followLast}$ sets in this manner [10].

A context-free grammar G is a quadruple (V, T, P, S) [27], where V is a finite set of nonterminal symbols, T is a finite set of terminal symbols, P is a finite set of productions of the form $V \to (V \cup T)^*$, and $S \in V$ is the start symbol. The language accepted by the grammar G is the set of words w from T^* satisfying $S \underset{G}{\overset{*}{\Longrightarrow}} w$, where $S \underset{G}{\overset{*}{\Longrightarrow}} w$ means that w is derivable from S (refer to [27] for more details). We say a nonterminal X is useful if there exists a word w such that $X \underset{G}{\overset{*}{\Longrightarrow}} w$. When all of the nonterminals in a production are useful, we say the production is valid.

3 Syntax for DREGs

In this section, we will first provide sound and complete inference systems for DREGs in Section 3.1, then show that DREGs can be defined by context-free grammars in Section 3.2.

3.1 Inference Systems for DREGs

The inference systems for DREGs is inspired by the following lemma.

Lemma 1 ([6, 25, 9])**.** *Let r be a regular expression.*

(1) $r = \varepsilon$*, or* $a \in \Sigma : r$ *is deterministic.*

(2) $r = r_1 | r_2 : r$ *is deterministic iff* r_1 *and* r_2 *are deterministic and* $\mathsf{First}(r_1) \cap \mathsf{First}(r_2) = \emptyset$*.*

(3) $r = r_1r_2$: *(A) If* $\varepsilon \in L(r_1)$*, then* r *is deterministic iff* r_1 *and* r_2 *are deterministic,* $\mathsf{First}(r_1) \cap \mathsf{First}(r_2) = \emptyset$*, and* $\mathsf{followLast}(r_1) \cap \mathsf{First}(r_2) = \emptyset$*.* *(B) If* $\varepsilon \notin L(r_1)$*, then* r *is deterministic iff* r_1 *and* r_2 *are deterministic, and* $\mathsf{followLast}(r_1) \cap \mathsf{First}(r_2) = \emptyset$*.*

(4) $r = r_1^+$: r *is deterministic iff* r_1 *is deterministic and the following condition holds:* $\forall x \in \mathsf{followLast}(r_1'), \forall y \in \mathsf{First}(r_1')$*, if* $x^\natural = y^\natural$ *then* $x = y$*.*

(5) $r = r_1?$: r *is deterministic iff* r_1 *is deterministic.*

This lemma gives a characterization of DREGs as well as an algorithm to decide whether a given regular expression is deterministic. Furthermore, we can also use it as a method to construct DREGs as shown in the following.

Note that the condition in case (4) is characterized by marked expressions, while the inference systems will only handle original expressions. So we first revise the above characterization such that the condition is on unmarked expressions as well. For that, we define a Boolean function $\mathcal{P}$ on unmarked expressions as follows.

Definition 2 ([9])**.** *Let* r *be a regular expression. The Boolean function* $\mathcal{P}(r)$ *is inductively defined as follows:*

$$\begin{aligned}
&\mathcal{P}(\varepsilon) = \mathcal{P}(a) = \mathsf{true} \quad a \in \Sigma\\
&\mathcal{P}(r_1 \mid r_2) = \mathcal{P}(r_1) \wedge \mathcal{P}(r_2) \wedge (\mathsf{followLast}(r_2) \cap \mathsf{First}(r_1) = \emptyset)\\
&\qquad\qquad \wedge (\mathsf{followLast}(r_1) \cap \mathsf{First}(r_2) = \emptyset)\\
&\mathcal{P}(r_1r_2) = (\neg\lambda(r_1) \wedge \neg\lambda(r_2) \wedge (\mathsf{followLast}(r_2) \cap \mathsf{First}(r_1) = \emptyset)) \vee\\
&\quad (\lambda(r_1) \wedge \neg\lambda(r_2) \wedge \mathcal{P}(r_2) \wedge (\mathsf{followLast}(r_2) \cap \mathsf{First}(r_1) = \emptyset)) \vee\\
&\quad (\neg\lambda(r_1) \wedge \lambda(r_2) \wedge \mathcal{P}(r_1) \wedge (\mathsf{followLast}(r_2) \cap \mathsf{First}(r_1) = \emptyset)\\
&\qquad\qquad \wedge (\mathsf{First}(r_1) \cap \mathsf{First}(r_2) = \emptyset)) \vee\\
&\quad (\lambda(r_1) \wedge \lambda(r_2) \wedge \mathcal{P}(r_1) \wedge \mathcal{P}(r_2) \wedge (\mathsf{followLast}(r_2) \cap \mathsf{First}(r_1) = \emptyset))\\
&\mathcal{P}(r_1^+) = \mathcal{P}(r_1?) = \mathcal{P}(r_1)
\end{aligned}$$

Intuitively, the function $\mathcal{P}(r)$ checks the following condition: whether there exists a symbol b and two distinct indexes i and j such that $b_i \in \mathsf{followLast}(r')$ and $b_j \in \mathsf{First}(r')$, *i.e.*, the condition in case (4) of Lemma 1: $\forall x \in \mathsf{followLast}(r')$, $\forall y \in \mathsf{First}(r')$, if $x^\natural = y^\natural$ then $x = y$. That is the following property.

Proposition 1 ([9])**.** *Given a DREG* r*,* $\mathcal{P}(r) = \mathsf{true}$ *iff the following condition holds:* $\forall x \in \mathsf{followLast}(r'), \forall y \in \mathsf{First}(r')$*, if* $x^\natural = y^\natural$ *then* $x = y$*.*

Based on this property, we can restate case (4) of Lemma 1 as follows:

Proposition 2. r^+ *is deterministic iff* r *is deterministic and* $\mathcal{P}(r) = \mathsf{true}$*.*

Using the properties above, we provide an inference system $\mathcal{D}$ for DREGs over Σ, which consists of the following rules (here $\vdash r$ means r is a DREG):

$$\text{(Empty)}\ \frac{}{\vdash \varepsilon} \qquad \text{(Const)}\ \frac{a \in \Sigma}{\vdash a}$$

$$\text{(Union)}\ \frac{\vdash r \quad \vdash s \quad \mathsf{First}(r) \cap \mathsf{First}(s) = \emptyset}{\vdash r \mid s}$$

$$\text{(ConcatA)}\ \frac{\vdash r\ \ \vdash s\ \neg\lambda(r)\ \mathsf{followLast}(r)\cap\mathsf{First}(s)=\emptyset}{\vdash r\cdot s}$$

$$\text{(ConcatB)}\ \frac{\vdash r\quad \vdash s\quad \lambda(r)\quad \mathsf{followLast}(r)\cap\mathsf{First}(s)=\emptyset \qquad \mathsf{First}(r)\cap\mathsf{First}(s)=\emptyset}{\vdash r\cdot s}$$

$$\text{(Plus)}\ \frac{\vdash r\ \ \mathcal{P}(r)}{\vdash r^{+}}\qquad \text{(Que)}\ \frac{\vdash r}{\vdash r?}$$

Each rule in $\mathcal{D}$ corresponds to one case in Lemma 1. Take the rule (ConcatA) as an example. If r and s are deterministic, $\lambda(r) = \mathsf{false}$ and $\mathsf{followLast}(r) \cap \mathsf{First}(s) = \emptyset$, then $r \cdot s$ is deterministic by the case $(3.B)$ in Lemma 1. We say r is derivable if there is a derivation tree in $\mathcal{D}$ whose root is r. If r is derivable in $\mathcal{D}$, r is clearly deterministic by Lemma 1. On the other hand, given a DREG r, we can construct for r a derivation tree, which is isomorphic to the structure of r. Thus r is derivable in $\mathcal{D}$. That is to say, the inference system $\mathcal{D}$ is sound and complete.

Theorem 1. *A regular expression r is deterministic iff r is derivable from $\mathcal{D}$.*

The inference system $\mathcal{D}$ can help users understand how to construct DREGs incrementally.

3.2 Context-free Grammars for DREGs

In this section, we will present grammars for DREGs, which are constructed by simulating the computations in the inference system $\mathcal{D}$.

Suppose that r is a DREG and r_1 is a sub-expression of r. Clearly r_1 is also deterministic. According to the inference system $\mathcal{D}$, if we replace r_1 in r by any DREG with the same values of First, $\mathsf{followLast}$, λ and $\mathcal{P}$ as r_1, then the resulting expression is still a DREG. This is the key property that enables us to construct the context-free grammars for DREGs.

Lemma 2. *Given a DREG r, if r_1 is a subexpression of r, and r_2 is a DREG which has the same values of* First, $\mathsf{followLast}$, *λ and $\mathcal{P}$ as r_1, then replacing r_1 by r_2 in r results in another DREG.*

Proof. Actually, we can prove a stronger result: replacing r_1 by r_2 in r results in a DREG having the same values of First, $\mathsf{followLast}$, λ and $\mathcal{P}$ as r. We prove it by induction on the structure of r.

$r = a,\ a \in \Sigma$: Then r_1 is r. When we replace r_1 by r_2, the resulting expression is r_2. Since r_2 is deterministic, and both r_1 and r_2 have the same First, $\mathsf{followLast}$, λ and $\mathcal{P}$, the conclusion follows.

$r = r_3|r_4$: Here we only show the case r_1 is a subexpression of r_3; the case that r_1 is a subexpression of r_4 can be proved similarly. Since r_1 is a subexpression of r_3, and r_1 and r_2 have the same First, $\mathsf{followLast}$, λ and $\mathcal{P}$, by the inductive hypothesis, we know that replacing r_1 by r_2 in r_3 results in a DREG, denoted by r_3', which has the same First, $\mathsf{followLast}$, λ and $\mathcal{P}$ as r_3. According to Lemma 1, and the computations of First, $\mathsf{followLast}$, λ and $\mathcal{P}$ (see Section 2 and Section 3.1), we know that $r_3'|r_4$ is also deterministic, and both $r_3'|r_4$ and $r_3|r_4$ have the same values of First, $\mathsf{followLast}$, λ and $\mathcal{P}$. Then the conclusion follows.

The other cases can be proved likely. □

Assume the alphabet of DREGs is $\Sigma = \{a_1, \dots, a_n\}$. To construct the grammars for DREGs, we first define a finite set $\mathbb{X}$ of nonterminal symbols in which each symbol is of the form $X^{S,R,\alpha,\beta}$, where $S, R \subseteq \Sigma$ and $\alpha, \beta \in \{\mathsf{true}, \mathsf{false}\}$. Intuitively, the nonterminal symbol $X^{S,R,\alpha,\beta}$ is intended to describe the set, denoted as $L(X^{S,R,\alpha,\beta})$, of DREGs whose $\mathsf{First}, \mathsf{followLast}, \lambda$ and $\mathcal{P}$ values are S, R, α, and β, respectively.

Thanks to Lemma 2, we can replace any subexpression r_1 in a DREG r by any DREG with the same First, $\mathsf{followLast}$, λ and $\mathcal{P}$ values. The set of all these possible replacers are exactly the set $L(X^{\mathsf{First}(r_1),\mathsf{followLast}(r_1),\lambda(r_1),\mathcal{P}(r_1)})$. Accordingly, if we replace subexpressions by nonterminals, then we can obtain "productions" for DREGs. This suggests us to construct the productions from the rules in the inference system $\mathcal{D}$ as follows: replacing DREGs by nonterminals. Moreover, from the definition of $X^{S,R,\alpha,\beta}$, we need to consider the computations of First, $\mathsf{followLast}$, λ and $\mathcal{P}$ when constructing grammars. We also need to ensure that the conditions in the rules in $\mathcal{D}$ must hold. In detail, we construct the grammar in the following manner: (1) there is a class of productions corresponding to each kind of rule in $\mathcal{D}$; (2) each class of productions should conform to the computations of First, $\mathsf{followLast}$, λ and $\mathcal{P}$; (3) nonterminals are selected carefully such that the related conditions in the rules of $\mathcal{D}$ hold. The grammar, denoted as G_d, is shown in Figure 1, where $\bigcup$ denotes a set of rules with the same left hand nonterminal, and the symbols (,), |, $\cdot$, + and ?, the alphabet symbols a_i ($i = 1, \dots, n$), and ε are the terminals of the grammars. Any nonterminal symbol can be the start symbol. For a fixed alphabet, the number of nonterminal symbols is finite, so is the number of productions in G_d.

To illustrate the construction of G_d, let us consider the Union case. The productions for this case can be constructed in two steps: (1) compute the possible values of First, $\mathsf{followLast}$, λ and $\mathcal{P}$ that the two operands of | can have; and (2) check the conditions to ensure determinism in $\mathcal{D}$.

Assume that the DREG r we want to construct is in the form of $r_1|r_2$ with $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\lambda(r) = \alpha$ and $\mathcal{P}(r) = \beta$ (*i.e.*, $X^{S,R,\alpha,\beta} \xRightarrow[G_d]{*} r$). First, let us compute the possible values $S_i, R_i, \alpha_i, \beta_i$ for r_i (*i.e.*, $X^{S_i,R_i,\alpha_i,\beta_i} \xRightarrow[G_d]{*} r_i$), where $i = 1, 2$. According to the computations of First, $\mathsf{followLast}$, λ, and $\mathcal{P}$, the following conditions must hold:
(1) $S = \mathsf{First}(r) = \mathsf{First}(r_1) \cup \mathsf{First}(r_2) = S_1 \cup S_2$;
(2) $R = \mathsf{followLast}(r) = \mathsf{followLast}(r_1) \cup \mathsf{followLast}(r_2) = R_1 \cup R_2$;
(3) $\alpha = \lambda(r) = \lambda(r_1) \vee \lambda(r_2) = \alpha_1 \vee \alpha_2$;
(4) $\beta = \mathcal{P}(r) = \mathcal{P}(r_1) \wedge \mathcal{P}(r_2) \wedge (\mathsf{followLast}(r_2) \cap \mathsf{First}(r_1) = \emptyset) \wedge (\mathsf{followLast}(r_1) \cap$

$$
\begin{array}{llll}
\underline{\text{Base}}: & X^{\{a_i\},\emptyset,\mathsf{false},\mathsf{true}} \to a_i & X^{\emptyset,\emptyset,\mathsf{true},\mathsf{true}} \to \varepsilon & \\
\underline{\text{Union}}: & X^{S,R,\alpha,\beta} \to & \bigcup\limits_{\Phi(S,R,\alpha,\beta,S_1,R_1,\alpha_1,\beta_1,S_2,R_2,\alpha_2,\beta_2)} & (X^{S_1,R_1,\alpha_1,\beta_1} \mid X^{S_2,R_2,\alpha_2,\beta_2}) \\
\underline{\text{Concat}}: & X^{S,R,\alpha,\beta} \to & \bigcup\limits_{\Theta(S,R,\alpha,\beta,S_1,R_1,\alpha_1,\beta_1,S_2,R_2,\alpha_2,\beta_2)} & (X^{S_1,R_1,\alpha_1,\beta_1} \cdot X^{S_2,R_2,\alpha_2,\beta_2}) \\
\underline{\text{Plus}}: & X^{S,R,\alpha,\mathsf{true}} \to & \bigcup\limits_{R_1 \cup S = R} X^{S,R_1,\alpha,\mathsf{true}} + & \\
\underline{\text{Que}}: & X^{S,R,\mathsf{true},\beta} \to & \bigcup\limits_{\alpha_1 \in \{\mathsf{true},\mathsf{false}\}} X^{S,R,\alpha_1,\beta}\ ? &
\end{array}
$$

where $i = 1,\ldots,n$, $\Phi(S,R,\alpha,\beta,S_1,R_1,\alpha_1,\beta_1,S_2,R_2,\alpha_2,\beta_2) \overset{def}{=} (S_1 \cup S_2 = S)$ and $(\mathsf{S}_1 \cap \mathsf{S}_2 = \emptyset)$ and $(\mathsf{R}_1 \cup \mathsf{R}_2 = \mathsf{R})$ and $(\alpha_1 \vee \alpha_2 = \alpha)$ and $(\beta_1 \wedge \beta_2 \wedge (\mathsf{S}_1 \cap \mathsf{R}_2 = \emptyset) \wedge (\mathsf{R}_1 \cap \mathsf{S}_2 = \emptyset) = \beta)$, and $\Theta(S,R,\alpha,\beta,S_1,R_1,\alpha_1,\beta_1,S_2,R_2,\alpha_2,\beta_2) \overset{def}{=} (R_1 \cap S_2 = \emptyset)$ and $(\alpha_1 \wedge \alpha_2 = \alpha)$ and $(\alpha_1 \wedge (\mathsf{S}_1 \cup \mathsf{S}_2 = \mathsf{S}) \wedge (\mathsf{S}_1 \cap \mathsf{S}_2 = \emptyset) \vee \neg\alpha_1 \wedge (\mathsf{S}_1 = \mathsf{S}))$ and $(\alpha_2 \wedge (\mathsf{R}_1 \cup \mathsf{S}_2 \cup \mathsf{R}_2 = \mathsf{R}) \vee \neg\alpha_2 \wedge (\mathsf{R}_2 = \mathsf{R}))$ and $(\beta = (\neg\alpha_1 \wedge \neg\alpha_2 \wedge (\mathsf{S}_1 \cap \mathsf{R}_2 = \emptyset)) \vee (\alpha_1 \wedge \neg\alpha_2 \wedge \beta_2 \wedge (\mathsf{S}_1 \cap \mathsf{R}_2 = \emptyset)) \vee (\neg\alpha_1 \wedge \alpha_2 \wedge \beta_1 \wedge (\mathsf{S}_1 \cap \mathsf{R}_2 = \emptyset) \wedge (\mathsf{S}_1 \cap \mathsf{S}_2 = \emptyset)) \vee (\alpha_1 \wedge \alpha_2 \wedge \beta_1 \wedge \beta_2 \wedge (\mathsf{S}_1 \cap \mathsf{R}_2 = \emptyset)))$.

Figure 1: Context-free grammars for DREGs

$\mathsf{First}(r_2) = \emptyset) = \beta_1 \wedge \beta_2 \wedge (R_2 \cap S_1 = \emptyset) \wedge (R_1 \cap S_2 = \emptyset)$.
Moreover, to ensure determinism, the condition of the rule (Union) in $\mathcal{D}$, *i.e.*, (5) $\mathsf{First}(r_1) \cap \mathsf{First}(r_2) = S_1 \cap S_2 = \emptyset$, must hold as well. In conclusion, the productions for DREGs of the form $r_1|r_2$ must satisfy Conditions (1)-(5), which form the definition of Φ.

Clearly, the grammars are context-free. Next, we study the correctness of these grammars: r is a DREG iff r is derivable from G_d. More precisely, we prove that $L(X^{S,R,\alpha,\beta})$ is intended to describe the correct language, which is shown by Lemma 3 and Lemma 4.

Lemma 3. *If* $X^{S,R,\alpha,\beta} \underset{G_d}{\overset{*}{\Longrightarrow}} r$*, then* r *is a DREG with* $\mathsf{First}(r) = S$*,* $\mathsf{followLast}(r) = R$*,* $\lambda(r) = \alpha$ *and* $\mathcal{P}(r) = \beta$*.*

Proof. We prove it by induction on the length of the derivation $X^{S,R,\alpha,\beta} \underset{G_d}{\overset{*}{\Longrightarrow}} r$. For convenience, $X^{S,R,\alpha,\beta} \underset{G_d}{\overset{*}{\Longrightarrow}} r$ is written as $X^{S,R,\alpha,\beta} \overset{*}{\Rightarrow} r$.

Base case : If the derivation is one-step, then $X^{S,R,\alpha,\beta} \overset{*}{\Rightarrow} r$ is in the form of $X^{\{a_i\},\emptyset,\mathsf{false},\mathsf{true}} \to a_i$ $(i = 1,\ldots,n)$ or $X^{\emptyset,\emptyset,\mathsf{true},\mathsf{true}} \to \varepsilon$. It is easy to see that the statement is true.

Inductive step: Suppose that the derivation takes $k+1$ steps $(k \geq 1)$, and for any derivation of no more than k steps, the statement is true. To simplify the presentation, we denote $X^{S,R,\alpha,\beta}$ by X, $X^{S_1,R_1,\alpha_1,\beta_1}$ by X_1 and $X^{S_2,R_2,\alpha_2,\beta_2}$ by X_2.

If the derivation is in the form $X \Rightarrow X_1 \mid X_2 \overset{*}{\Rightarrow} r$, $r = r_1 \mid r_2$, $X_1 \overset{*}{\Rightarrow} r_1$ and $X_2 \overset{*}{\Rightarrow} r_2$. By the inductive hypothesis, we know that (1) $X_1 \overset{*}{\Rightarrow} r_1$ and $X_2 \overset{*}{\Rightarrow} r_2$ take no more than k steps; (2) r_1 and r_2 are DREGs; (3) $\mathsf{First}(r_1) = S_1$, $\mathsf{followLast}(r_1) = R_1$, $\lambda(r_1) = \alpha_1$, $\mathcal{P}(r_1) = \beta_1$, $\mathsf{First}(r_2) = S_2$,

$\mathsf{followLast}(r_2) = R_2$, $\lambda(r_2) = \alpha_2$ and $\mathcal{P}(r_2) = \beta_2$. And from the definition of the union rules, we have $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$, $R_1 \cup R_2 = R$, $\alpha_1 \vee \alpha_2 = \alpha$ and $(\beta_1 \wedge \beta_2 \wedge (S_1 \cap R_2 = \emptyset) \wedge (R_1 \cap S_2 = \emptyset)) = \beta$. Because r_1, r_2 are deterministic and $\mathsf{First}(r_1) \cap \mathsf{First}(r_2) = S_1 \cap S_2 = \emptyset$, r is a DREG from Lemma 1. Hence $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\lambda(r) = \alpha$ and $\mathcal{P}(r) = \beta$ from the computations of First, $\mathsf{followLast}$, λ, and $\mathcal{P}$.

If the first step of the derivation uses other rules, we can prove similarly. □

Lemma 4. *If r is a DREG with $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\lambda(r) = \alpha$ and $\mathcal{P}(r) = \beta$, then $X^{S,R,\alpha,\beta} \overset{*}{\underset{G_d}{\Longrightarrow}} r$.*

Proof. We prove it by induction on the structure of r. The cases for $r = \varepsilon$ or $a, a \in \Sigma$ are obvious.

For simplicity, we use X, X_1 and X_2 to denote $X^{\mathsf{First}(r),\mathsf{followLast}(r),\lambda(r),\mathcal{P}(r)}$, $X^{\mathsf{First}(r_1),\mathsf{followLast}(r_1),\lambda(r_1),\mathcal{P}(r_1)}$ and $X^{\mathsf{First}(r_2),\mathsf{followLast}(r_2),\lambda(r_2),\mathcal{P}(r_2)}$, respectively, where r, r_1 and r_2 are expressions. $X^{S,R,\alpha,\beta} \overset{*}{\underset{G_d}{\Longrightarrow}} r$ is written as $X^{S,R,\alpha,\beta} \overset{*}{\Rightarrow} r$.

If $r = r_1 \mid r_2$, from r is deterministic, we have that both r_1 and r_2 are also deterministic. By the inductive hypothesis, r_1 and r_2 are derivable from G_d. Then from Lemma 3, we know that $X_1 \overset{*}{\Rightarrow} r_1$ and $X_2 \overset{*}{\Rightarrow} r_2$. From the computations of λ and $\mathcal{P}$, and the fact that r is a DREG, we know that $\mathsf{First}(r) = \mathsf{First}(r_1) \cup \mathsf{First}(r_2)$, $\mathsf{followLast}(r) = \mathsf{followLast}(r_1) \cup \mathsf{followLast}(r_2)$, $\mathsf{First}(r_1) \cap \mathsf{First}(r_2) = \emptyset$, $\lambda(r) = \lambda(r_1) \vee \lambda(r_2)$, $(\mathcal{P}(r_1) \wedge \mathcal{P}(r_2) \wedge (\mathsf{First}(r_1) \cap \mathsf{followLast}(r_2) = \emptyset) \wedge (\mathsf{followLast}(r_1) \cap \mathsf{First}(r_2) = \emptyset)) = \mathcal{P}(r)$. Therefore, $X \Rightarrow X_1 \mid X_2 \overset{*}{\Rightarrow} r_1|r_2$. That is, $X \overset{*}{\Rightarrow} r$. Hence r is derivable from G_d.

We can prove the other cases similarly. □

Based on the lemmas above, we can conclude that

Theorem 2. *DREGs can be defined by context-free grammars.*

Proof. We only need to show that $L(X^{S,R,\alpha,\beta})$ is the intended language. That is to show: (1) The strings generated by G_d are DREGs (Lemma 3); and (2) All DREGs can be generated by G_d (Lemma 4). □

Using the pumping lemma [27] for regular languages and the parenthesis matching, we can prove that DREGs cannot be defined by regular grammars.

Proposition 3. *DREGs cannot be defined by regular grammars.*

Proof. The proof is based on the pumping lemma for regular languages [27]. Given an alphabet $\Sigma = \{a_1, \ldots, a_n\}$, let L denote the set of DREGs over Σ. Suppose that L can be defined by a regular grammar G, namely, $L = L(G)$, and its corresponding integer in the pumping lemma is m. Let us take $\underbrace{((\ldots((}_{l} a_1 a_1 \underbrace{)a_1)\ldots)a_1)}_{2l-1} \in L$ such that $m \leq l$. According to the pumping lemma, the substring y must consist entirely of (s. Assume that $|y| = k \geq 1$. Clearly, the string obtained for $i = 0$ is $w_0 = \underbrace{((\ldots(}_{l-k} a_1 a_1 \underbrace{)\ldots)a_1)}_{2l-1}$, which is clearly not deterministic, since the parentheses are not matched. This contradicts to the pumping lemma. Therefore, the set of DREGs is not regular. That is, DREGs cannot be defined by a regular grammar. □

4 Valid Productions

In this section, we will study the validity of productions, namely, to remove the useless symbols in the grammar.

Let us consider a general nonterminal $X^{S,R,\alpha,\beta}$: $X^{S,R,\alpha,\beta}$ is useful if there exists a DREG r such that $X^{S,R,\alpha,\beta} \overset{*}{\underset{G_d}{\Longrightarrow}} r$. Obviously, if $S = \emptyset$, then only the empty string ε satisfies this condition, and thus we have $R = \emptyset$, $\lambda = \mathsf{true}$, and $\beta = \mathsf{true}$. Moreover, when $S \cap R = \emptyset$, by Proposition 1, we have that $\beta = \mathsf{true}$. These two conditions are clearly the necessity condition for $X^{S,R,\alpha,\beta}$ being useful. Rather surprise, although they specify conditions for two extreme cases, we find that these two conditions are also the sufficiency condition. Since we can show that if the two conditions hold, then there exists a DREG r such that $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\lambda(r) = \alpha$, and $\mathcal{P}(r) = \beta$.

Lemma 5. *Given S, R, α, and β, the nonterminal symbol $X^{S,R,\alpha,\beta}$ is useful if and only if the following two conditions hold: (1) $(S = \emptyset) \to ((R = \emptyset) \wedge (\alpha = true) \wedge (\beta = true))$; (2) $(S \cap R = \emptyset) \to (\beta = true)$.*

Proof. ($\Leftarrow$) We will show that when the conditions hold, we can construct a DREG r such that $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\lambda(r) = \alpha$ and $\mathcal{P}(r) = \beta$. Then from the proof of Lemma 4, we know that $r \in L(X^{S,R,\alpha,\beta})$. That is, $X^{S,R,\alpha,\beta}$ is useful.

When $S \cap R \neq \emptyset$ and $S \neq \emptyset$, the expression r is constructed as follows. Suppose $S \cap R = \{a_1, \ldots, a_n\}$, $S = \{a_1, \ldots, a_n, b_1, \ldots, b_{m_1}\}$, and $R = \{a_1, \ldots, a_n, c_1, \ldots, c_{m_2}\}$. We need to consider the following cases:

- If $\alpha = 1$ and $\beta = 1$, then $r = \varepsilon|(a_1^+|\ldots|a_n^+|b_1|\ldots|b_{m_1}) \cdot (\varepsilon|c_1|\ldots|c_{m_2})$. We can verify that r is deterministic, $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\alpha = 1$, and $\beta = 1$.

- If $\alpha = 1$ and $\beta = 0$, then $r = \varepsilon|(a_1|\ldots|a_n|b_1|\ldots|b_{m_1}) \cdot (\varepsilon|a_1|\ldots|a_n|c_1|\ldots|c_{m_2})$. We can verify that r is deterministic, $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\alpha = 1$, and $\beta = 0$.

- If $\alpha = 0$ and $\beta = 1$, then $r = (a_1^+|\ldots|a_n^+|b_1|\ldots|b_{m_1}) \cdot (\varepsilon|c_1|\ldots|c_{m_2})$. We can verify that r is deterministic, $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\alpha = 0$, and $\beta = 1$.

- If $\alpha = 0$ and $\beta = 0$, then $r = (a_1|\ldots|a_n|b_1|\ldots|b_{m_1}) \cdot (\varepsilon|a_1|\ldots|a_n|c_1|\ldots|c_{m_2})$. We can verify that r is deterministic, $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\alpha = 0$, and $\beta = 0$.

When $S \cap R = \emptyset$ and $S \neq \emptyset$, by the hypothesis, we have $\beta = 1$. Then the expression r is constructed as follows. Let $S = \{b_1, \dots, b_{m_1}\}$ and $R = \{c_1, \dots, c_{m_2}\}$. Similarly, we consider the following cases:

- If $\alpha = 1$, then $r = \varepsilon|(b_1|\dots|b_{m_1}) \cdot (\varepsilon|c_1|\dots|c_{m_2})$. We can verify that $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\alpha = 1$, and $\beta = 1$.

- If $\alpha = 0$, then $r = (b_1|\dots|b_{m_1}) \cdot (\varepsilon|c_1|\dots|c_{m_2})$. We can verify that $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\alpha = 0$, and $\beta = 1$.

When $S = \emptyset$, by the hypothesis, we have that $R = \emptyset$, $\alpha = 1$, and $\beta = 1$. Then the expression $r = \varepsilon$ satisfies that $\mathsf{First}(r) = \emptyset$, $\mathsf{followLast}(r) = \emptyset$, $\alpha = 1$, and $\beta = 1$.

Therefore if the conditions hold, then $X^{S,R,\alpha,\beta}$ is useful.

($\Rightarrow$) Suppose that $S \cap R = \emptyset$. Then the following condition holds: $\forall x \in \mathsf{followLast}(r'), \forall y \in \mathsf{First}(r')$, if $x^\natural = y^\natural$ then $x = y$. By Proposition 1, we have that $\beta = 1$.

When $S = \emptyset$, because $X^{S,R,\alpha,\beta}$ is useful, by Lemma 3 there exists a DREG r such that $\mathsf{First}(r) = \emptyset$. Since we assume that regular expressions are not $\emptyset$, from $\mathsf{First}(r) = \emptyset$ we know that $L(r) = \{\varepsilon\}$. Hence $\mathsf{followLast}(r) = \emptyset$, $\alpha = 1$, and $\beta = 1$.

This completes the proof. □

Next, we study the effectiveness of the lemma. Given an alphabet Σ, there are $2^{|\Sigma|}$ different First and $\mathsf{followLast}$ sets, so there are $2^{|\Sigma|} \cdot 2^{|\Sigma|} \cdot 2 \cdot 2 = 2^{2|\Sigma|+2}$ different nonterminal symbols. Each production uses at most three nonterminals, then the number of possible productions is in $O(5 \times 2^{6|\Sigma|+6})$. Table 1 lists the numbers of productions in G_d over small alphabets, which increases rapidly. Note that permutation of symbols in First and $\mathsf{followLast}$ sets does not matter.

Let us consider the number of valid productions (*i.e.*, the ones without useless nonterminals). We use G_o to denote such optimized grammars with only valid productions. Table 1 also lists the numbers of productions in G_o over small alphabets. From the table, we can see that the numbers of productions in G_o are much fewer than the ones in G_d. According to Lemma 5, we know that G_o denotes the same language as G_d. This demonstrates the effectiveness of Lemma 5.

Table 1: $|G_d|$ and $|G_o|$ over small alphabets

$\|\Sigma\|$	1	2	3	4	5
$\|G_d\|$	208	3297	43698	542387	6553772
$\|G_o\|$	46	815	14904	240481	3520010

At last, we show that for a given alphabet Σ, there are valid productions in all the five classes of productions in G_d. The base case is trivial, since these productions can generate words in one step. We show that there are valid productions in the other classes in the following. Actually the conditions in the productions are corresponding to the conditions for an expression to be deterministic and the processes of computing First, $\mathsf{followLast}$, λ and $\mathcal{P}$. Let us consider the following DREGs $r_1 = \varepsilon|a$, $r_2 = aa$, $r_3 = a^+$ and $r_4 = a?$ for any symbol $a \in \Sigma$. For each expression, we try to construct one valid production for each class of productions. The productions we constructed are as follows: $(P1)$ $X_2 \to X_0|X_1$, $(P2)$ $X_1 \to X_1 \cdot X_1$, $(P3)$ $X_3 \to X_1+$, and $(P4)$ $X_2 \to X_1?$, where X_0, X_1, X_2, and X_3 stand for $X^{\emptyset,\emptyset,\mathsf{true},\mathsf{true}}$, $X^{\{a\},\emptyset,\mathsf{false},\mathsf{true}}$, $X^{\{a\},\emptyset,\mathsf{true},\mathsf{true}}$, and $X^{\{a\},\{a\},\mathsf{false},\mathsf{true}}$, respectively. Since X_is are useful, all these productions are valid.

5 Random Generation of DREGs

Due to the fact that there were no syntax definitions for DREGs before, by now there are no tools for automatic generating DREGs. Nevertheless, in Section 3.2 we have shown that DREGs can be defined by context-free grammars. In this section, we further show that these context-free grammars can be used to automatically generate DREGs. Our aim is to generate DREGs randomly and efficiently.

5.1 Random Generation Algorithm

With the grammars presented in Section 3.2, a naive idea is to construct the whole grammar for a given alphabet first and then use the existing methods [28, 29, 30, 31, 32] to randomly generate sentences from this constructed grammar. However, due to the facts that the grammars are too large to construct fully (see Section 3.2 and Table 1), and that existing methods require some pre-processings of the whole grammar, the existing methods become inefficient and thus are not suitable for the grammars of DREGs. In existing methods, the pre-processings are to keep the information needed for randomness, thus contribute to make the random generation efficient. Hence the key problem is how to make the random generation efficient without the construction and the pre-processings of the whole grammar. Thanks to the information that the symbols in the grammars of DREGs take (*i.e.*, the parameters of the nonterminal symbols), we can take full advantage of this information to make the random generation efficient without constructing the whole grammar.

Our solution consists of the following ideas:

- Construct the valid productions only when they are needed to avoid the construction of the whole grammar. Thanks to Lemma 5, we can construct the valid productions directly. Otherwise, we need to construct the whole grammar to check the validity of each production. This could lead to an inefficient generation.

- Impose a conservative length condition to make the generation efficient: if the length of the generating sentence exceeds a given length, the generation terminates with a failure. This is similar to most existing work, but different in that existing work requires the length of the generated sentence is exactly the given one, while we require the length is no longer than the given one. This is because that we would like to avoid the pre-processings of the whole grammar that existing work require. Moreover, for other structure requirements of the generated DREGs, such as concerning the occurrences of regular expression operators, the generation may become more difficult, such as the pre-processings cannot be avoided.

- Employ a length control mechanism to improve the efficiency of generation. In detail, we (over-)estimate the possible (minimum) length l_e for the generating sentence, which can be solved by using the information of the symbols. Then we compare l_e with the given length l. If $l_e \geq l$, the generation starts to select the productions with small First and followLast randomly. As a result, the First and followLast become smaller and smaller until the Base case of the grammar. Therefore, the generation can terminate soon.

Algorithm 1 Random Generation Algorithm

```
Input: the length $l$ and the alphabet $\Sigma$
Output: a sentence $s$ s.t. $|s| \leq l$ or failure
 1: generate a useful non-terminal $X_s$ randomly and put it into Stack $stack$
 2: set current sentence $s_c \leftarrow \epsilon$, length $l_c \leftarrow 0$
 3: set length control off $LC \leftarrow$ false
 4: while $stack$ is nonempty and $l_c \leq l$ do
 5:   $X \leftarrow$ pop $stack$
 6:   if $X$ is terminal then
 7:     $s_c \leftarrow s_c \cdot X,\ \ l_c \leftarrow l_c + 1$ if $X \in \Sigma$
 8:   else
 9:     if $LC$ is false then
10:       $l_r \leftarrow estLen(stack),\ \ LC \leftarrow (l_c + l_r > l)$
11:     end if
12:     if $LC$ is false then
13:       construct a production $p$ for $X$ randomly
14:     else
15:       construct $p$ randomly s.t. $\forall X' \in rhs(p).\ X'.S \subset X.S$ and $X'.R \subset X.R$
16:     end if
17:     if the construction fails then return failure
18:     end if
19:     push $rhs(p)$ into $stack$ reversely
20:   end if
21: end while
22: if $stack$ is empty then return $s_c$
23: else return failure end if
```

The random generation algorithm is shown in Algorithm 1, which takes an alphabet Σ and a length l as input, and returns a sentence with length no longer than l or a failure. The algorithm starts with a stack with a random useful non-terminal (Line 1). In other words, the algorithm starts with two random subsets $S, R \subseteq \Sigma$ and two random Boolean values α, β such that $X^{S,R,\alpha,\beta}$ is useful, *i.e.* $X^{S,R,\alpha,\beta}$ satisfies the conditions in Lemma 5. And the algorithm initializes the current generating sentence s_c as empty string ϵ and its length l_c as 0 respectively (Line 2), and set the length control mechanism off (Line 3). Then the algorithm proceeds each (top) symbol in the stack until either the stack is empty or the length of the current generating sentence exceeds the required one ($l_c \geq l$) (Lines 4-21). In detail, if the proceeding symbol X is a terminal, then the algorithm concatenates it with the current sentence and increases the length by 1 if it belongs to Σ (Lines 6-7). Otherwise, the algorithm uses the length function *estLen* to estimate a possible length l_r that the current stack can generate, and compares it with the remaining length (*i.e.*, the required length l minus the current one l_c) to trigger on the length control mechanism (Line 10). Followed up, the algorithm tries to construct a production p for the symbol X following the rules in Figure 1 (Lines 12-16) depending on the length control. If the length control mechanism is off, p is selected uniformly at random from all valid productions of X (Line 13); otherwise, p comes randomly from those productions satisfying that each non-terminal symbol X' on the right-hand side has a smaller First set S and a smaller followLast set R than X (Line 15). If the construction fails, the algorithm terminates with a failure (Lines $17-18$), otherwise, it pushes all the symbols on the right-hand side of p into the stack reversely (Line 19), and continues with the next possible symbol. Finally, if the stack is empty, the algorithm returns the generated sentence (Line 22); otherwise, it returns a failure (Line 23).

5.2 Length Estimation

In this section we present how to define the length estimating function *estLen* to make the random generation efficient without any pre-processings of the whole grammar.

First, similar to existing work [44, 41], let us consider the problem of estimating the minimum length for the sentences generated from a symbol. Clearly, the length of sentence generated from a terminal symbol is fixed (*i.e.*, 1), while it is variant from a non-terminal symbol. L. Zheng and D. Wu have presented a solution to calculate the minimum length and to construct the corresponding sentence that can be generated from a non-terminal symbol [44]. However, this solution requires some pre-processings of the whole grammar as well, so it is not applicable here.

The key difficulty lies in how to decide whether there exists a sentence s generated from a given non-terminal symbol X for a given length l such that $|s| = l$, based on which, the smallest l can be found. Thanks to the information (*i.e.*, the First set S and the followLast set R) that a non-terminal symbol takes, we can conclude that any sentence s generated from $X^{S,R,\alpha,\beta}$ satisfies that $|s| \geq |S \cup R|$, since $S \cup R \subseteq occur(s)$.

Table 2: DREGs with length $|S \cup R|$ and $|S \cup R| + 1$

S	R	α	β = true	β = false
$\emptyset$	$\emptyset$	true	ε	useless
$\emptyset$	$\emptyset$	false	useless	useless
$\emptyset$	P_2	true	useless	useless
$\emptyset$	P_2	false	useless	useless
P_1	$\emptyset$	true	$(a_1 \vert \ldots \vert a_m)?$	useless
P_1	$\emptyset$	false	$a_1 \vert \ldots \vert a_m$	useless
P_1	P_2	true	$((a_1 \vert \ldots \vert a_n)(\varepsilon \vert c_1 \vert \ldots \vert c_p))^*$	useless
P_1	P_2	false	$((a_1 \vert \ldots \vert a_n)(\varepsilon \vert c_1 \vert \ldots \vert c_p))^+$	useless
P_3	P_3	true	$(b_1 \vert \ldots \vert b_n)^*$	$\varepsilon \vert b_1 \cdot b_1^+ \vert b_2^+ \vert \ldots \vert b_n^+$
P_3	P_3	false	$(b_1 \vert \ldots \vert b_n)^+$	$b_1 \cdot b_1^+ \vert b_2^+ \vert \ldots \vert b_n^+$
P_3	$P_2 \cup P_3$	true	$((b_1 \vert \ldots \vert b_n)(\varepsilon \vert c_1 \vert \ldots \vert c_p))^*$	$\varepsilon \vert (b_1 \cdot b_1^+ \vert b_2^+ \vert \ldots \vert b_n^+)(\varepsilon \vert c_1 \vert \ldots \vert c_p)$
P_3	$P_2 \cup P_3$	false	$((b_1 \vert \ldots \vert b_n)(\varepsilon \vert c_1 \vert \ldots \vert c_p))^+$	$(b_1 \cdot b_1^+ \vert b_2^+ \vert \ldots \vert b_n^+)(\varepsilon \vert c_1 \vert \ldots \vert c_p)$
$P_1 \cup P_3$	P_3	true	$(\varepsilon \vert a_1 \vert \ldots \vert a_m)(\varepsilon \vert b_1 \vert \ldots \vert b_n)$	$(\varepsilon \vert a_1 \vert \ldots \vert a_m)(\varepsilon \vert b_1 \cdot b_1^+ \vert b_2 \vert \ldots \vert b_n)$
$P_1 \cup P_3$	P_3	false	$(\varepsilon \vert a_1 \vert \ldots \vert a_m)(b_1 \vert \ldots \vert b_n)^+$	$(\varepsilon \vert a_1 \vert \ldots \vert a_m)(b_1 \cdot b_1^+ \vert b_2^+ \vert \ldots \vert b_n^+)$
$P_1 \cup P_3$	$P_2 \cup P_3$	true	$\varepsilon \vert (a_1 \vert \ldots \vert a_m \vert b_1^+ \vert \ldots \vert b_n^+)(\varepsilon \vert c_1 \vert \ldots \vert c_p)$	$\varepsilon \vert (a_1 \vert \ldots \vert a_m \vert b_1 \cdot b_1^+ \vert b_2 \vert \ldots \vert b_n^+)(\varepsilon \vert c_1 \vert \ldots \vert c_p)$
$P_1 \cup P_3$	$P_2 \cup P_3$	false	$(a_1 \vert \ldots \vert a_m \vert b_1^+ \vert \ldots \vert b_n^+)(\varepsilon \vert c_1 \vert \ldots \vert c_p)$	$(a_1 \vert \ldots \vert a_m \vert b_1 \cdot b_1^+ \vert \ldots \vert b_n^+)(\varepsilon \vert c_1 \vert \ldots \vert c_p)$

Lemma 6. *Given S, R, α, and β, then for each $s \in L(X^{S,R,\alpha,\beta})$, $|s| \geq |S \cup R|$.*

Proof. Due to $S \subseteq occur(s)$ and $R \subseteq occur(s)$, we can get $|S \cup R| \leq |s|$. □

Lemma 6 gives us a solution, but what we want is the minimum or the one which is very close to the minimum. For that, we try to find whether there exists a sentence whose length is very close to $|S \cup R|$. Table 2 gives the constructed DREGs, indicating that there exists a DREG either with length $|S \cup R|$ when β is true or with length $|S \cup R| + 1$ when β is false for a useful non-terminal $X^{S,R,\alpha,\beta}$, where $P_1 = \{a_1, \ldots, a_m\}$, $P_2 = \{c_1, \ldots, c_p\}$ and $P_3 = \{b_1, \ldots, b_n\}$ such that P_is are pairwise disjoint. So we declare that the solution given by Lemma 6 is reasonable.

Let us take the minimum length $|R \cup S|$ as a reference for our length estimating function, and assume that the possible length exceeds the required length. So the length control mechanism is triggered on. From then on, we have to lead the generation to terminating with a sentence whose length does not exceed the given one. Therefore, we have to select the productions that lead to the minimum sentence due to the minimum reference. But it is a pity that the whole grammar is "unknown", so is the minimum sentence. Even worse, this selection could lose randomness, even we use the sentences presented in Table 2 randomly.

As mentioned above, instead of the minimum sentence, we still select the productions randomly, but from the ones with smaller First and followLast. This smaller selection enables us to select the productions randomly again and to lead the generation to terminating. To satisfy the length condition, the length generated by the smaller selection should be smaller than the remaining length. In other words, the remaining length should be as long as possible. Therefore, the earlier to trigger on the length control mechanise, the higher the success rate. On the other hand, we would like to generate the sentences whose lengths are as close to the given one as possible. Conversely, this requirement requires the generation to start the length control mechanise as late as possible. Again, the problem is how to define the length function to trigger the length control mechanism on *properly*, or to estimate the possible length of the sentences generated by the smaller selection.

Similarly, let us take the possible *minimum* length by the smaller selection as a reference. Consider the production cases of a non-terminate symbol $X^{S,R,\alpha,\beta}$. For **Plus** and **Que**, there is only symbol $X^{S_1,R_1,\alpha,\beta}$ and $|S_1 \cup R_1| = |S \cup R|$. So the possible minimum length is almost the same. While for **Union** and **Concat**, there are two symbols $X^{S_1,R_1,\alpha,\beta}$ and $X^{S_2,R_2,\alpha,\beta}$. Under the smaller selection, $S = S_1 \cup S_2$, $S_1 \cap S_2 = \emptyset$ and $R_1 \cup R_2 = R$. So $|S \cup R| \leq |S_1 \cup R_1| + |S_2 \cup R_2| < |S_1 \cup R| + |S_2 \cup R| = |S_1| + |R| + |S_2| + |R| = |S| + 2 * |R|$. Hence, the possible minimum length is on the increase if **Union** and **Concat** are applied. To make matters worse, these two cases may be applied on continuously ($|S|$ times at most) until the **Base** case (*i.e.* $|S_i| = 1$). So the possible minimum length can be $|S| + |R| + |S| * |R|$ at worst. Conservatively, this worst minimum length can be used as the length function *estLen*.

Lemma 7. *Given an non-terminate $X^{S,R,\alpha,\beta}$, the possible minimum length for the DREGs generated from $X^{S,R,\alpha,\beta}$ under the smaller selection is $|S \cup R| + |S| * |R|$ at worse.*

Proof. This can be proved by induction on the derivation trees of expressions.

- **Base:** Trivially.
- **Union:** According to the definition of Φ, we have $S = S_1 \cup S_2$, $S_1 \cap S_2 = \emptyset$ and $R_1 \cup R_2 = R$. Let the minimum lengths of DREGs from $X^{S,R,\alpha,\beta}, X^{S_1,R_1,\alpha_1,\beta_1}, X^{S_2,R_2,\alpha_2,\beta_2}$ are m, m_1, m_2, respectively.

$$\begin{aligned} m &= m_1 + m_2 \\ &\leq |S_1 \cup R_1| + |S_1| * |R_1| + |S_2 \cup R_2| + |S_2| * |R_2| \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(By induction)} \\ &\leq |S_1 \cup R| + |S_1| * |R| + |S_2 \cup R| + |S_2| * |R| \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad (R_i \subseteq R) \\ &= |S_1 \cup S_2 \cup R| + |S_1 \cup S_2| * |R| \quad (S_1 \cap S_2 = \emptyset) \\ &= |S \cup R| + |S| * |R| \quad (S = S_1 \cup S_2) \end{aligned}$$

- **Concat:** Due to the smaller selection, we have $|S_i| < |S|$ and $|R_i| < |R|$. This makes us have to select the productions such that $\alpha_i = \mathsf{true}$[1], which yields $S = S_1 \cup S_2$, $S_1 \cap S_2 = \emptyset$ and $R_1 \cup S_2 \cup R_2 = R$. Then similar to the case of **Union**, we can prove the minimum length is $\leq |S \cup R| + |S| * |R|$ at most.
- **Plus:** Let the minimum lengths of DREGs from $X^{S,R,\alpha,\mathsf{true}}$ and $X^{S,R_1,\alpha,\mathsf{true}}$ are m and m', respectively. By induction $m' \leq |S \cup R_1| + |S| * |R_1|$. Clearly, we have $m = m'$. Thus $m \leq |S \cup R_1| + |S| * |R_1| \leq |S \cup R| + |S| * |R|$ since $R_1 \subseteq R$.

[1]In our implementation, if such productions do not exist, we relax the conditions of smaller selection as $|S_i| \leq |S|$ and $|R_i| \leq |R|$ such that α_i can be false, rather than returning a failure.

- **Que:** Similar to the case of **Plus**.

□

While in practice, we also try some other choice, such as the average of the minimum length and the worst minimum length (*i.e.*, $(|S \cup R| + 0.5 * |S| * |R|)$). More details are presented in Section 6.

6 Experiments

We have implemented a prototype to construct the DREG grammars and to generate DREGs from these grammar in Java. Using our prototype, we conducted a series of experiments to evaluate our solution. Firstly, we conducted experiments to see whether the DREG grammars can be constructed and further be used easily in practice. Secondly, we conducted experiments to test the performance of our random generation algorithm on different estimating functions. Thirdly, to test the efficientness further, we also conducted experiments to compare our algorithm with the RE Algorithm, *i.e.*, the algorithm to generate regular expressions first and then select the deterministic ones. Finally, to see the usefulness of our generator, we applied the generated DREGs in an inclusion checker for DREGs.

All the experiments are conducted on a machine with Intel core I5 CPU and 4GB RAM, running Ubuntu 14.04.

6.1 DREG Grammar Construction

Given an alphabet Σ, constructing the grammar is easy, just by constructing all the possible valid productions according to the rules presented in Figure 1. Thanks to Lemma 5, those invalid productions can be thrown away directly, without any additional check on the whole grammar. Table 3 shows the experimental results for the optimized grammars with small alphabets, where $|G_o|$ denotes the number of productions in optimized grammar G_o, $Time$ and Avg_T denote the constructing time for the grammar and the average time for each production respectively, and *Stored* denotes the size of the file that stores the grammar. Although the grammars can be constructed easily, the result shows that the time and space needed by the grammars are exponential on $|\Sigma|$. This is because, as explained in Section 4, the number of productions is $O(5 \times 2^{6|\Sigma|+6})$.

Due to the large scale, it seems that these grammars cannot be easily used in practice. For example, we have tried to generate DREGs using Dong's algorithm [32], which is a linear algorithm to enumerate sentences from a CFG and can be used as a sentence generator provided with a randomizer [41]. But we only succeed for the grammar whose symbol size is smaller than 7, due to the large memory required by the maintaining of the whole grammar and the useful information for generation.

However, in some cases, only some productions for a specific non-terminal symbol are required, rather than the whole grammars. For example, when

Table 3: Grammar construction for given alphabet Σ

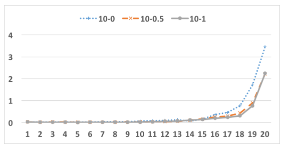
$\|\Sigma\|$	$\|G_o\|$	$Time$	$Stored$	Avg_T
1	46	0.033ms	1.69k	0.711us
2	815	0.817ms	35.2k	1.002us
3	14904	7.276ms	726k	0.488us
4	240481	119.113ms	12.6M	0.495us
5	3520010	2323.929ms	203M	0.660us
6	48369939	28506.531ms	2.98G	0.589us
7	638241628	396844.845ms	42.7G	0.622us

using these grammars for sentence generation (*e.g.*, our random generation) only one production is required for each non-terminal symbol at a time. Moreover, the experimental result also shows that the average time for constructing a production is very low (almost smaller than $1us$). This enables us to only construct the productions efficiently when they are needed.

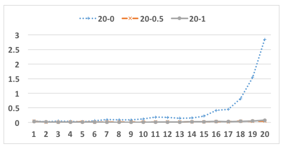
6.2 DREG Generation

As we discuss in Section 5.2, the length estimating function *estLen* affects the sentence generation. Hence we conduct some experiments to evaluate our random generation algorithm: how the algorithm perform on different estimating functions for grammars with different alphabet sizes. For that, we generate 100 sentences randomly with length no longer than $10, 20, 50$ from the grammars, whose alphabet size ranges from 1 to 20, taking the worst minimum length (*i.e.*, $|S \cup R| + |S| * |R|$), the minimum length (*i.e.*, $|S \cup R|$), and their average (*i.e.*, $|S \cup R| + 0.5 * |S| * |R|$) as the length function *estLen*, respectively. During generating, we collect the total time, the average length, and the failure number.

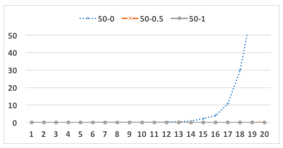
The experimental results are shown in Figure 2, where the horizontal axises of all the figures represent the sizes of the grammar's alphabets, the vertical axises of Figures 2(a) - 2(c), Figures 2(d) - 2(f), and Figures 2(g) -2(i) represent the total times in seconds, the average length radios with respect to the required length and the failure numbers, respectively, and "*l*-*n*" means that the required length is l and the length function *estLen* is $|S \cup R| + n * |S| * |R|$. The results show that the total times costed by most DREG generations, except the case of 50-0 for the grammars with the alphabet size larger than 15, are in several seconds, indicating that our algorithm can generate DREGs efficiently. In particular, taking $|S \cup R| + |S| * |R|$ and $|S \cup R| + 0.5 * |S| * |R|$ as the length function, the cost time could be in 1s, even in 0.1s when the required length is large. Moreover, from Figures 2(a)-2(c), we can see that (i) the higher the length function, the less (better) the cost time; and (ii) the cost time would not increase rapidly (almost the same) as the alphabet size increases. The reason is that with a higher estimated length, the length control mechanism can be triggered on earlier and there are more "space" for the smaller random selection.



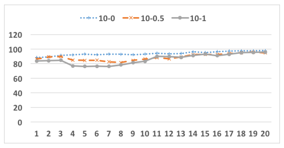
(a) Total Time for $l \leq 10$



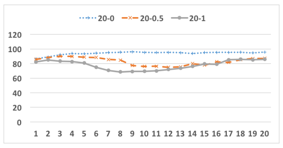
(b) Total Time for $l \leq 20$



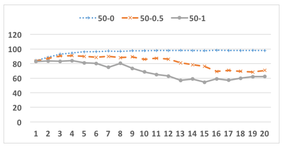
(c) Total Time for $l \leq 50$



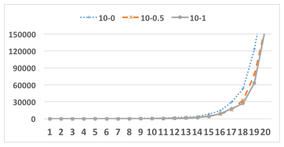
(d) Aver. Len. for $l \leq 10$



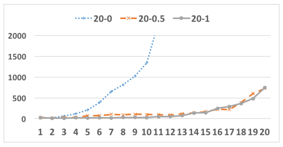
(e) Aver. Len. for $l \leq 20$



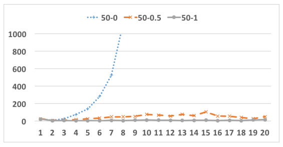
(f) Aver. Len. for $l \leq 50$



(g) Fail. Num. for $l \leq 10$



(h) Fail. Num. for $l \leq 20$



(i) Fail. Num. for $l \leq 50$

Figure 2: Results for different length estimations

While considering the actual lengths of the generated sentences, from Figures 2(d)-2(f), we can see that the average lengths of most DREG generations account for more than 60% of the required ones. In particular, taking the minimum length as the the length function performs best such that the average length of the generated DREGs is quite close to the required one. Moreover, different from the cost time, the higher the length function, the smaller (worse) the average length.

Finally, due to the same reason as for the cost time, the higher the estimated length, the less the failure number. Indeed, the failure number contributes to the cost time. Figure 2(i) shows that with $|S \cup R| + |S| * |R|$ and $|S \cup R| + 0.5 * |S| * |R|$ for the length function, the failure number is no more than 100 to generate sentences with length no longer than 50.

As a conclusion, we suggest the average minimum length (*i.e.*, $|S \cup R| + 0.5 * |S| * |R|$) as the estimated length function in practice, taking both the cost time and the average length into account.

Finally, we would like to see how our algorithm perform to generate *long* sentences from *large* grammars as well. So we try to generate 100 sentences with length no longer than 500 for grammars whose alphabet size ranges from 21 to 27[2]. Table 4 shows the experimental results, where $|\Sigma|$ denotes the size of alphabet Σ, TTime denotes the total time in seconds for the random generation, Failure denotes the failure number during the generation, and ALen denotes the average length of the generated sentences, respectively. The results shows that our algorithm is still effective to generate *long* sentences from *large* grammars:

[2]From 28, due to the Java heap space, the generation fails

Table 4: Results for $Len \leq 500$

$\|\Sigma\|$	TTime	Failure	ALen	$\|\Sigma\|$	TTime	Failure	ALen
21	8.540	1	357.58	22	9.085	1	360.49
23	15.943	1	359.42	24	34.888	1	350.48
25	64.419	0	350.21	26	132.174	0	346.57
27	583.298	0	335.8				

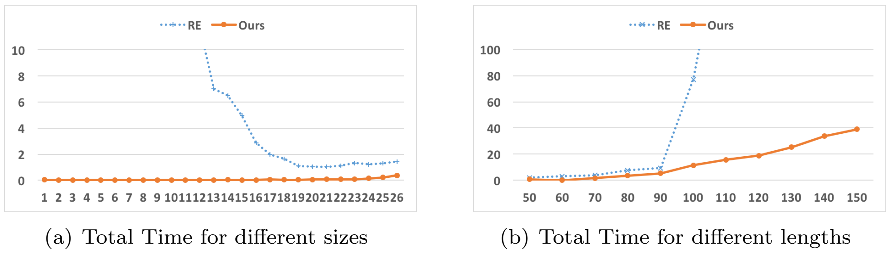
generating a DREG with length no longer than 500 from a grammar with alphabet size 27 is about $5.833s$ on average. We also found that the cost time rises as the alphabet size increases. However, thanks to the estimated function we take, the failure number is very few (almost 0), which indicates that the cost time is almost caused by the essential generation. Moreover, the average lengths still account for more than 60% of the required lengths.

6.3 Comparison with the RE Algorithm

In this section, to evaluate the efficiency of our algorithm, we compare our algorithm with the RE one, namely, to generate regular expressions (RE for short) first using a random sentence generator for RE and then select those deterministic ones [9]. To avoid the time caused by the pre-proceedings, we take the simplest random generation algorithm, namely Hanford's algorithm [34], as the RE generator. We also adopt the length control mechanism [44] to ensure the termination. In detail, the RE generator takes as input the grammar for REs: $R \to a_i \mid R\,R \mid R\;o\;R \mid R\,+ \mid R\;? \mid \varepsilon$, where a_i represents any symbol in alphabet and o denotes the alternation. The RE generator starts with the nonterminal R and selects a rule for each nonterminal randomly during generation. Meanwhile, the RE generator counts the current length of the generated sentential form. If the current length exceeds the given one, then the RE generator starts to select the terminal rules $R \to a_i$ randomly.

First, we try to generate sentences of the same length from grammars with different sizes to compare our algorithm and the RE one. To do that, from the grammars with alphabet size from 1 to 26, we first use our algorithm to generate 100 sentences with length no longer than 50, and then use the RE one to generate 100 sentences whose lengths are exactly the same as the corresponding average length of the sentences generated by our algorithm. The results are shown in Figure 3(a), where the horizontal axis represents the sizes of the grammar's alphabets and the vertical axis represents the total times in seconds costed by our algorithm and the RE one. The results show that our algorithm is much more efficient than the RE one, especially for the grammars with small alphabet. There are two factors that make the RE algorithm inefficient[3]: one is the very

[3]If some other algorithms, such as Hickey and Cohen's algorithm [35], were used, the pre-proceedings could be another factor.



(a) Total Time for different sizes

(b) Total Time for different lengths

Figure 3: Comparison between ours and the RE algorithm

low ratio of DREGs in regular expressions (*e.g*, to generate 100 DREGs with length 50 from grammar with alphabet size 26 needs about 8618 REs with a radio 1.2%) and the other is the determinism checking, which has to be done for every RE. While our algorithm generates DREGs directly such that these two factors are absent, yielding an efficient generator. Moreover, if the required length 50, rather than the average length from our algorithm, was used for the RE algorithm, then it would cost much more time (see the following experiment), even that the algorithm could not terminate in a day for the grammars with small alphabet.

Next, we also compare our algorithm with the RE one by generating sentences with different lengths from the same grammar, for example, the one with alphabet size 26. First, we use our algorithm to generate 100 sentences with length no longer than a given length, ranging from 50 to 150, and then proceeds as the above experiment. Figure 3(b) shows the results, where the horizontal axis represents the given length restricted on sentences and the vertical axis is the same as Figure 3(a). The results show that as the given length increases, the times costed by the RE one increase rapidly, such that it cannot generate a DREG with length bigger than 100 in an hour. In particular, when the given length is much longer than the alphabet size, it cannot generate a DREG in a day. This is mainly due to the probability that the generated regular expressions are deterministic is decreased very quickly, as the given length increases. On the contrary, thanks to the estimated function, our algorithm still performs well: (i) it generates the required DREGs with length no longer than 50 in 1 second; and (ii) the cost time seems linear on the required length.

The above results show that RE generators are not feasible for generating DREGs, and our algorithm is quite efficient for generating DREGs. Due to the simplicity (fewer pre-proceedings) and the randomness (the very low ratio of DREGs in RE) of the RE algorithm we select, we can conclude that our algorithm cannot be replaced by any RE generation algorithm.

6.4 Application of DREG Generator

As mentioned in Section 1, an automatic generator of DREGs is indispensable in practice. In this section, we present the application of our DREG generator

Table 5: Results for the inclusion checker

RLen	ALen	LMax	LMin	GTime	CTime
50	39.504	50	19	0.820	0.231
100	72.245	100	25	10.294	0.360
150	122.278	150	34	35.106	0.493
200	159.594	200	54	43.301	0.546

in the inclusion checker for DREGs in [47].

The inclusion checker presented in [47] implemented two inclusion algorithms, both of which take two DREGs r_1, r_2 as input and return true if $r_1 \subseteq r_2$ and false otherwise. To evaluate the inclusion checker, many DREGs of different lengths are required. However, due to the inefficient generation (*i.e.*, the RE generator), the inclusion checker are evaluated only on DREGs with length no more than 30. Here we use our DREG generator to generate long DREGs from the grammar with alphabet size 26 and use these DREGs to evaluate the inclusion checker further. In detail, similar to [47], we first generate 100 pairs of DREGs with length no longer than 50, 100, 150 and 200 respectively, then apply the inclusion checker on these 100 pairs of generated DREGs, wherein we account the average length of the generated DREGs, the generated time, and the checking time. Table 5 gives the experimental results, where RLen and ALen denote the required length and the average length of the generated sentences respectively, LMax and LMin denote the maximum and the minimum of the sentence lengths respectively, GTime and CTime denote the times in seconds costed by the generation and by the checking respectively. The results shows that the inclusion checker still works effectively on DREGs with length no longer than 200.

This experiment demonstrates that our DREGs generator can help to evaluate the inclusion checker. We believe our generator can be used in some other applications which require automatic generation of DREGs, and thus is useful in practice.

7 Conclusion

In this paper, we gave syntactic grammars for DREGs, and showed that the class of DREGs is context-free. Each production of the grammars is of the form $X \to a_i|\varepsilon, a_i \in \Sigma$ (base) or $X \to Y\ uo$ (unary operators) or $X \to Y\ bo\ Z$ (binary operators), where *uo* denotes the unary operators and *bo* the binary operators. Each nonterminal symbol is of the form $X^{S,R,\alpha,\beta}$ which defines the language $L(X^{S,R,\alpha,\beta}) = \{r \in \text{DREGs} \mid \mathsf{First}(r) = S, \mathsf{followLast}(r) = R, \lambda(r) = \alpha, \mathcal{P}(r) = \beta\}$. We showed that $L(X^{S,R,\alpha,\beta})$ is the intended language.

We further designed a random generator for DREGs. The generator does not construct the whole grammar, instead it constructs productions only when they are needed. Further, it imposes a conservative length condition and a length control mechanism. These ensure the efficiency of the generator. Experiments showed that the generator is efficient and useful in practice.

In the future, there are several work to be done. (1) Further studies of grammar constructions as started in this paper. It is possible to find other ways to construct the grammars, for example by using the continuing property [10], in the hope to find some smaller grammars, which remains as a future work. (2) Extending regular expressions by other operators. In this paper we are considering standard regular expressions. To include other commonly used operators, such as counting and interleaving, will be useful. (3) Optimizations, which will be quite useful for the context-free grammars for DREGs. We will examine other optimization rules to further reduce the number of productions in the grammars. (4) Other applications of the grammars. For example we will further develop a tool for helping the user to write DREG expressions, in which the use of grammars is necessary. (5) Examining practical subclasses of DREGs which may have simpler grammars than DREGs such that the use of the grammars can be more efficient. (6) Investigating the pre-processings to make guarantees about the distribution on sentences of length n by taking the symmetric similarity of DREG grammars into account. (7) Algorithms for random generation of XSDs in which the random generation algorithm for DREGs must be used.

Acknowledgements

The idea of giving a syntax for DREGs and the initial context-free rules were developed while the first author was visiting Frank Neven. Since Neven changed research topic soon, he decided that he will not continue doing this work and we can continue the work and use the ideas created during the visit freely. So we thank Frank Neven for this. We also thank Wim Martens for fruitful discussion about an early version of this paper. This work is supported by the National Natural Science Foundation of China under Grants No. 61472405 and 61502308, and is partially funded by Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University.

8 Reference

References

[1] (2006). Extensible Markup Language (XML) 1.1. `http://www.w3.org/TR/xml11`.

[2] (2005). World Wide Web Consortium. `http://www.w3.org/wiki/UniqueParticleAttribution`.

[3] Losemann, K. and Martens, W. (2012) The complexity of evaluating path expressions in SPARQL. *PODS*, pp. 101–112. ACM.

[4] Huang, X., Bao, Z., Davidson, S. B., Milo, T., and Yuan, X. (2015) Answering regular path queries on workflow provenance. *ICDE*, pp. 375–386. IEEE.

[5] Brüggemann-Klein, A. (1993) Regular expressions into finite automata. *Theor. Comput. Sci.*, **120**, 197–213.

[6] Brüggemann-Klein, A. and Wood, D. (1998) One-unambiguous regular languages. *Inf. Comput.*, **142**, 182–206.

[7] Gelade, W., Gyssens, M., and Martens, W. (2012) Regular expressions with counting: weak versus strong determinism. *SIAM J. Comput.*, **41**, 160–190.

[8] Sperberg-McQueen, C. M. (2004). Notes on finite state automata with counters. `http://www.w3.org/XML/2004/05/msm-cfa.html`.

[9] Chen, H. and Lu, P. (2011) Assisting the design of XML Schema: diagnosing nondeterministic content models. *APWeb*, pp. 301–312.

[10] Chen, H. and Lu, P. (2015) Checking determinism of regular expressions with counting. *Inf. Comput.*, **241**, 302–320.

[11] Losemann, K., Martens, W., and Niewerth, M. (2016) Closure properties and descriptional complexity of deterministic regular expressions. *Theor. Comput. Sci.*, **627**, 54–70.

[12] Bex, G. J., Gelade, W., Martens, W., and Neven, F. (2009) Simplifying XML schema: effortless handling of nondeterministic regular expressions. *SIGMOD*, New York, NY, USA, pp. 731–744. ACM.

[13] Groz, B. and Maneth, S. (2017) Efficient testing and matching of deterministic regular expressions. *J. Comput. Syst. Sci.*, **89**, 372–399.

[14] Lu, P., Peng, F., Chen, H., and Zheng, L. (2015) Deciding determinism of unary languages. *Inf. Comput.*, **245**, 181–196.

[15] Lu, P., Peng, F., and Chen, H. (2013) Deciding determinism of unary languages is coNP-complete. *DLT*, pp. 350–361.

[16] Lu, P., Bremer, J., and Chen, H. (2015) Deciding determinism of regular languages. *Theory Comput. Syst.*, **57**, 97–139.

[17] Latte, M. and Niewerth, M. (2015) Definability by weakly deterministic regular expressions with counters is decidable. *MFCS*, pp. 369–381.

[18] Czerwinski, W., David, C., Losemann, K., and Martens, W. (2017) Deciding definability by deterministic regular expressions. *J. Comput. Syst. Sci.*, **88**, 75–89.

[19] Peng, F., Chen, H., and Mou, X. (2015) Deterministic regular expressions with interleaving. *ICTAC*, pp. 203–220.

[20] Ahonen, H. (1996) Disambiguation of SGML content models. *PODP*, pp. 27–37.

[21] Bex, G. J., Neven, F., Schwentick, T., and Tuyls, K. (2006) Inference of concise DTDs from XML data. *VLDB*, pp. 115–126. VLDB Endowment.

[22] Bex, G. J., Neven, F., and Vansummeren, S. (2007) Inferring XML Schema definitions from XML data. *VLDB*, pp. 998–1009. VLDB Endowment.

[23] Bex, G. J., Gelade, W., Neven, F., and Vansummeren, S. (2010) Learning deterministic regular expressions for the inference of schemas from XML data. *ACM Trans. Web*, **4**, 14:1–14:32.

[24] Freydenberger, D. D. and Kötzing, T. (2013) Fast learning of restricted regular expressions and DTDs. *ICDT*, pp. 45–56.

[25] Kilpeläinen, P. (2011) Checking determinism of XML Schema content models in optimal time. *Inf. Syst.*, **36**, 596–617.

[26] Kilpeläinen, P. and Tuhkanen, R. (2007) One-unambiguity of regular expressions with numeric occurrence indicators. *Inf. Comput.*, **205**, 890–916.

[27] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2007) *Introduction to automata theory, languages, and computation, third edition.* Addison-Wesley.

[28] Arnold, D. B. and Sleep, M. R. (1980) Uniform random generation of balanced parenthesis strings. *ACM Trans. Program. Lang. Syst.*, **2**, 122–128.

[29] Mairson, H. G. (1994) Generating words in a context-free language uniformly at random. *Inf. Process. Lett.*, **49**, 95–99.

[30] McKenzie, B. (1997). Generating strings at random from a context free grammar.

[31] Denise, A., Roques, O., and Termier, M. (2000) Random generation of words of context-free languages according to the frequencies of letters. *Mathematics and computer science. Algorithms, trees, combinatorics and probabilities. Proceedings of the colloquium, September 18–20, 2000*, pp. 113–125. Birkhäuser Basel.

[32] Dong, Y. (2009) Linear algorithm for lexicographic enumeration of CFG parse trees. *Science in China (Series F - Information Science)*, **52**, 1177–1202.

[33] Gelade, W. and Neven, F. (2012) Succinctness of the complement and intersection of regular expressions. *ACM Trans. Comput. Logic*, **13**, 1–19.

[34] Hanford, K. V. (1970) Automatic generation of test cases. *IBM Systems Journal*, **9**, 242–257.

[35] Hickey, T. and Cohen, J. (1983) Uniform random generation of strings in a context-free language. *SIAM J. Comput.*, **12**, 645–655.

[36] Gore, V., Jerrum, M., Kannan, S., Sweedyk, Z., and Mahaney, S. R. (1997) A quasi-polynomial-time algorithm for sampling words from a context-free language. *Inf. Comput.*, **134**, 59–74.

[37] Bertoni, A., Goldwurm, M., and Santini, M. (2001) Random generation for finitely ambiguous context-free languages. *RAIRO-Theor. Inf. Appl.*, **35**, 499–512.

[38] Gardy, D. and Ponty, Y. (2010) Weighted random generation of context-free languages: Analysis of collisions in random urn occupancy models. *GASCOM-8th conference on random generation of combinatorial structures.*

[39] Lorenz, W. A. and Ponty, Y. (2013) Non-redundant random generation algorithms for weighted context-free grammars. *Theor. Comput. Sci.*, **502**, 177–194.

[40] Héam, P.-C. and Masson, C. (2011) A random testing approach using pushdown automata. *TAP*, pp. 119–133. Springer.

[41] Xu, Z., Zheng, L., and Chen, H. (2010) A toolkit for generating sentences from context-free grammars. *SEFM*, pp. 118–122.

[42] Purdom, P. (1972) A sentence generator for testing parsers. *BIT Numerical Mathematics*, **12**, 366–375.

[43] Shen, Y. and Chen, H. (2005) Sentence generation based on context-dependent rule coverage. *Computer Engineering and Applications*, **41**, 96–100. In Chinese.

[44] Zheng, L. and Wu, D. (2009) A sentence generation algorithm for testing grammars. *COMPSAC*, pp. 130–135.

[45] Mäkinen, E. (1997) On lexicographic enumeration of regular and context-free languages. *Acta Cybern.*, **13**, 55–61.

[46] Antonopoulos, T., Geerts, F., Martens, W., and Neven, F. (2013) Generating, sampling and counting subclasses of regular tree languages. *Theory of Computing Systems*, **52**, 542–585.

[47] Chen, H. and Chen, L. (2008) Inclusion test algorithms for one-unambiguous regular expressions. *ICTAC*, Berlin, Heidelberg, pp. 96–110. Springer Berlin Heidelberg.