Towards an Effective Syntax and a Generator for DREGs

Zhiwu Xu[1], Ping Lu[2] and Haiming Chen[3]

[1]College of Computer Science and Software Engineering, Shenzhen University
[2]Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing, 100191
[3]State Key Laboratory of Computer Science, Institute of Software Chinese Academy of Sciences, Beijing 100190, China
Email: chm@ios.ac.cn

Deterministic regular expressions are a core part of XML Schema and used in other applications. But unlike regular expressions, deterministic regular expressions do not have a simple syntax, instead they are defined in a semantic manner. Moreover, not every regular expression can be rewritten to an equivalent deterministic regular expression. These properties of deterministic regular expressions put a burden on the user to develop XML Schema Definitions and to use deterministic regular expressions. In this paper, we propose a syntax for deterministic standard regular expressions (DREGs), and prove that the syntax of DREGs is context-free. Based on the context-free grammars for DREGs, we further design a generator for DREGs, which can generate DREGs randomly, and be used in applications associated with DREGs, *e.g.*, benchmarking a validator for DTD or XML Schema, and inclusion checking of DTD and XML Schema. Experimental results demonstrate the efficiency and usefulness of the generator.

Keywords: Deterministic regular expressions, Grammars, Properties, Optimizations, Generation

1. INTRODUCTION

Deterministic regular expressions 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 r is deterministic, we need to verify that for every two words w_1, w_2 in the language $L(r)$ represented by the regular expression, whether w_1 or w_2 satisfy some condition (see Definition 2.1). Moreover, deterministic regular expressions are strictly less expressive than regular expressions, *i.e.,* not every regular expression can be rewritten to an equivalent deterministic regular expression. These put a burden on the user to develop XML Schema Definitions (XSDs) and use deterministic regular expressions. Deterministic regular expressions 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].

Grammars for DREGs and their properties. In this paper, we propose a syntax for deterministic standard regular expressions (we call deterministic regular expressions obtained from standard regular expressions as deterministic standard regular expressions, denoted as DREGs) in terms of a set of parameters. We show that the syntax of DREGs is context-free, *i.e.*, the class of DREGs can be characterized by context-free grammars. Although the syntax of regular expressions is context-free, some proper subsets of regular expressions may not be context-free. For example, both $\{a^n b^n c^n \mid n \in \mathcal{N}\}$ and $\{abc^n \mid n \in \mathcal{N}\}$ are proper subsets of regular expressions, but they are context-sensitive and regular respectively, where $\mathcal{N}$ represents the set of all positive natural numbers.

Although there has been a lot of work on constructing DREGs for users from different perspectives, e.g., [6, 9, 10, 12, 19, 20, 21, 22, 23], there does not exist any simple syntax for DREGs. Such a syntax is important for the applications using DREGs, *e.g.*, benchmarking a validator for DTD or XML Schema, and the inclusion testing of DTD and XML Schema. In this paper, we will consider how to give a context-free grammar generating all DREGs over a given alphabet.

The construction of our syntax is achieved by building inference systems for DREGs, which are then used for giving grammars for DREGs. In detail, 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 in the following manner: (1) each nonterminal symbol has a set of parameters (*e.g.*, a set S and a set R), intending to describe the set of DREGs that satisfy these parameters

(*e.g.*, the First set is S and the followLast set is R); and (2) each production satisfies a set of equations of these parameters from left and right nonterminals. In this way, we have actually defined a class of grammars, and the grammars that we construct to characterize the DREGs over the alphabets of the same size are isomorphic.

We prove that the grammars exactly define DREGs, and are context-free, which indicates that the syntax of DREGs is context-free. Furthermore, by mapping DREGs to the Dyck languages, we prove that the classes of DREGs cannot be defined by regular grammars.

To effectively use the grammars, we further figure out some necessary and sufficient conditions for valid productions (see Section 2 for its definition), which can significantly reduce the number of valid productions in the grammars. For example, by using these conditions, the number of productions for the alphabet size 3 drops from 43698 to 14904, which 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 can be used in many applications such as the testing, experiments of programs having input of DREG types, data experiments needed plenty of DREG sentences, and synthetic benchmark for programs that work with DTDs or XML Schema. For example, suppose that we want to benchmark a validator for DTD or XML Schema. To this end, we can generate a set of schemas, and then automatically generate XML documents in the languages of these schemas using the work from [24]. Using these schemas and documents, we can test the scalability of the validator by increasing the sizes of the schemas and XML documents. Because of lacking of a simple syntax for DREGs before, one possible way to generate test cases is as follows: first use a sentence generator for regular expressions, then decide whether the generated expressions are deterministic. Due to the fact that the ratio of DREGs in regular expressions is quite small (*e.g.*, the ratio of DREGs is less than 1% for regular expressions of size 50 with alphabet size 40 [9]), within a certain period of time the user may not be able to obtain enough DREG sentences as needed. Furthermore, if the user also has requirements on the structure of DREGs, such as sizes of DREGs, the generation will be more difficult.

Based on the context-free grammars for DREGs, we design a generator to generate DREGs efficiently. Due to that the grammars are too large to fully construct (see Section 3.2 and Table 5), most of the existing methods are not suitable here. Thanks to the parameters in the nonterminal symbols of the grammars for DREGs, we can take full advantage of this information to efficiently generate DREGs. 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) select the productions of a nonterminal uniformly; (3) impose a conservative size condition to make the generation efficient; (4) employ a size control mechanism to improve the efficiency of the generation.

Experiments of the generator. Finally, we implemented a prototype of the generator in *Java* and 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.

- We propose a syntax for DREGs (Section 3), and prove that the syntax of DREGs is context-free. Meanwhile, we also show that the class of DREGs cannot be defined by regular grammars.
- 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.
- Based on the grammars, we further design and implement a random generation algorithm for DREGs (Section 5), in which productions are selected uniformly and only constructed when they are needed. It further imposes a conservative size condition and a size control mechanism to ensure efficiency.
- We experimentally evaluate the generator (Section 6). The results show that 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 show that repeatedly generating arbitrary regular expressions until we obtain a DREG is not a feasible option to generate DREGs.

Related work. The related work can be categorized as follows.

Deterministic regular expressions. To decide whether a standard expression is deterministic, Brüggemann-Klein [5] gave an algorithm with time complexity $O(|\Sigma_E||E|)$, which is based on the Glushkov automaton of the expression. For expressions with counting, Kilpeläinen [25] presented an $O(|\Sigma_E||E|)$ algorithm by examining the marked expression. Based on [9] and [25], Chen and Lu also gave an $O(|\Sigma_E||E|)$ algorithm for checking determinism of expressions with counting [10]. Groz and Maneth [13] gave the first $O(|E|)$ algorithm for checking determinism of standard regular expressions and expressions with counting. Peng *et al.* [18] gave an $O(|\Sigma_E||E|)$ algorithm for checking determinism of expressions with interleaving.

One may think the automata, *e.g.*, the Glushkov automata, defined for DREGs can be seen as a syntactic definition for DREGs. But we think they are more likely to be used for checking the determinism of a regular

expression, rather than defining the set of DREGs. Therefore, we do not wish to automatically generate DREGs by generating Glushkov automata.

There has been a lot of work on constructing DREGs for users, *e.g.*, [6, 19, 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 (*i.e.*, definable by a DREG). 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 [19] proposed an algorithm to construct approximate DREGs. Bex *et al.* [12] had also developed algorithms to construct approximate regular expressions. But here the languages are approximated, and solutions are needed to represent 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.* [20] 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 [21]. In a later paper [22], they provided algorithms to learn deterministic k-occurrence regular expressions, which are based on the Hidden Markov Model. Recently, Freydenberger and Kötzing [23] gave a linear time algorithm to infer SOREs and CHAREs from samples. Moreover, their algorithm generates the optimal SOREs and CHAREs [23].

The definability problem of DREGs is to express a given regular expression by an equivalent DREG, which was proved to be PSPACE-complete [12, 17, 15]. Latte and Niewerth [16] showed that one can decide in 2EXPSPACE whether a standard regular expression is equivalent to any deterministic regular expression with counting, whereas the exact complexity is still open. Lu *et al.* [14] proved that whether a given standard regular expression on a unary alphabet has an equivalent DREG is coNP-complete. Losemann *et al.* [11] studied the descriptional complexity of DREGs and showed that exponential blow-ups cannot be avoided. Gelade *et al.* [26] examined the descriptional complexity of intersection for DREGs and proved that the exponential bound on intersection is tight for DREGs.

Sentence Generation. Hanford [27] 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 [29] presented two algorithms to generate sentences of length n uniformly at random from a general context-free grammar. Follow-up research proposed several uniform random generation algorithms to improve either the time and space bounds [30, 31] or the pre-processing [32]. According to the frequencies of letters, Denise *et al.* [33] proposed two other uniform random generation algorithms. Some researchers considered other types of grammars, such as Bertoni *et al.* [34] took the ambiguity into account, while Ponty *et al.* [35, 36] considered the weighted context-free grammars. Besides, Héam and Masson [37] presented a uniform random generation algorithm using pushdown automata. Dong's enumeration algorithm [38] can be used as a random generation algorithm as well [39]. However, to make the generation efficient and/or to ensure the uniformity, all these algorithms require some pre-processing 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-processing. Moreover, our random generation consider a weak size condition (no larger than a given size) and the uniformity on productions of a same nonterminal symbol, so it may lose the uniformity on sentences of size n.

Besides random generation, some studies adopted other strategies, like coverage criteria [40, 41], length control [42], and sentence enumeration [38, 43].

For random generation of XSD, Antonopoulos *et al.* [24] initiated the work of uniform XSD generation by developing an algorithm for random generation of k-occurrence automata (k-OAs) for content 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 [24].

2. DEFINITIONS

Let Σ be an alphabet of symbols. The set of all finite words over Σ is denoted by Σ^*. A (standard) regular expression over Σ is ε, or $a \in \Sigma$, or 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, where ε represents the empty word. Notice that r^* (the *Kleene star*) is an abbreviation of $\varepsilon \mid r^+$, and we will not consider regular expressions like r^* in the following. To avoid unnecessary parentheses it is assumed that the plus and the question mark have the highest priority, concatenation follows, and union has the lowest priority. If there is no ambiguity, then parentheses may be omitted. For example, $(a?) \mid (b(c^+))$ can be written as $a? \mid bc^+$. We write $r_1 \mid r_2 \mid \ldots \mid r_n$ short for $(\ldots(r_1 \mid r_2)\ldots \mid r_n)$, where $n \geq 2$. The *size* of r is the

number of symbols in Σ occurring in r, also called the *alphabetic width* of r, denoted by $|r|$. The set of symbols occurring in r is denoted by $sym(r)$. For example, $|(a \mid b)^+ab(a \mid b)| = 6$ and $sym((a \mid b)^+ab(a \mid b)) = \{a, b\}$. Since in this paper we only consider standard regular expressions, in the following we will directly use regular expressions to represent standard regular expressions.

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 $\lambda(r) = \mathsf{false}$ otherwise. The computation of λ is listed as follows [5]:

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

Next we define deterministic regular expressions (DREGs for short). 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. Without loss of generality, we use the positions as the marked subscripts. For example, a marking of the expression $(a \mid b)^+ab(a \mid b)$ is $(a_1 \mid b_2)^+a_3b_4(a_5 \mid b_6)$. The marking of an expression r is denoted by $r^\sharp$. Accordingly the result of dropping the subscripts from a marked expression r is denoted by $r^\natural$. Then we have $(r^\sharp)^\natural = r$. We extend the notation for words in an obvious way. By using these notations, we can define DREGs as follows.

Definition 2.1 ([6]). *An expression r is deterministic if it satisfies the following condition: for any two words $uxv, uyw \in L(r^\sharp)$ with $|x| = |y| = 1$, if $x \neq y$, then $x^\natural \neq y^\natural$ holds. A regular language is deterministic if it can be denoted by some deterministic expression.*

For example, the expression $(a \mid \varepsilon)((c \mid d)^+ \mid \varepsilon)(a \mid \varepsilon)$ is not deterministic, since two words $a_1, a_4 \in L((a_1 \mid \varepsilon)((c_2 \mid d_3)^+ \mid \varepsilon)(a_4 \mid \varepsilon))$, and $a_1 \neq a_4$, 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 cannot define all 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:

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

Intuitively, $\mathsf{First}(r)$ contains all the first symbols of words in $L(r)$, while $\mathsf{followLast}(r)$ contains all the symbols b which can be appended to a nonempty word v in $L(r)$ to form another word vbw in $L(r)$. The computations for First and followLast can be found in [5, 25, 10]. But for the completeness of the paper, we list them here.

The computation of First can be done as follows [5] :

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

The computation of followLast on marked regular expressions and DREGs is listed as follows [25, 10]:

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

Note that the computation of followLast above cannot be applied on general regular expressions [10]. One reason is due to the union case $r + s$: for a word v belonging to one subexpression r, an appended word vbw may belong to the other subexpression s. Take the expression $a \mid aa$ for example. Following the computation above, we have $\mathsf{followLast}(a \mid aa) = \emptyset$, while it should be $\{a\}$ according to the definition.

A context-free grammar (CFG) G is a quadruple (V, T, P, S) [44], 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 $S \overset{*}{\underset{G}{\Longrightarrow}} w$, where $S \overset{*}{\underset{G}{\Longrightarrow}} w$ means that w is derivable from S (refer to [44] for more details), and the subscript G can be dropped if it is clear from the context. We say that a nonterminal X is useful if there exists a word w from T^* such that $X \overset{*}{\underset{G}{\Longrightarrow}} w$. When all 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, and then show that DREGs can be defined by context-free grammars in Section 3.2.

3.1. Inference System for DREGs

Our inference systems for DREGs are inspired by the determinism checking of regular expressions in [6, 25, 9], which is formalized as the following lemma.

Lemma 3.1 ([6, 25, 9]). *A regular expression r is deterministic if and only if*

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

(2) $r = r_1 \mid r_2 : r_1$ *and* r_2 *are deterministic and* $\mathsf{First}(r_1) \cap \mathsf{First}(r_2) = \emptyset$.
(3) $r = r_1 r_2 : r_1$ *and* r_2 *are deterministic,* $\mathsf{followLast}(r_1) \cap \mathsf{First}(r_2) = \emptyset$, *and if* $\varepsilon \in L(r_1)$ *then* $\mathsf{First}(r_1) \cap \mathsf{First}(r_2) = \emptyset$.
(4) $r = r_1^+ : r_1$ *is deterministic and* $\forall x \in \mathsf{followLast}(r_1^\sharp), \forall y \in \mathsf{First}(r_1^\sharp)$, *if* $x^\natural = y^\natural$ *then* $x = y$.
(5) $r = r_1? : r_1$ *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 here we need original expressions. So we first revise the characterization above such that the condition is on unmarked expressions as well. For that, we adopt a Boolean function $\mathcal{P}$ on unmarked expressions defined in [9].

DEFINITION 3.1 ([9]). *Let* r *be a regular expression. The Boolean function* $\mathcal{P}(r)$ *is inductively defined as follows:*

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

Intuitively, the function $\mathcal{P}(r)$ checks whether there exists a symbol b and two distinct indexes i and j such that $b_i \in \mathsf{followLast}(r^\sharp)$ and $b_j \in \mathsf{First}(r^\sharp)$. More specifically, $\mathcal{P}(r)$ ensures the condition in case (4) of Lemma 3.1: $\forall x \in \mathsf{followLast}(r^\sharp)$ and $\forall y \in \mathsf{First}(r^\sharp)$, if $x^\natural = y^\natural$ then $x = y$. This is formalized in the following proposition.

PROPOSITION 3.1 ([9]). *Given a DREG* r, $\mathcal{P}(r) = \mathsf{true}$ *if and only if the following condition holds:* $\forall x \in \mathsf{followLast}(r^\sharp), \forall y \in \mathsf{First}(r^\sharp)$, *if* $x^\natural = y^\natural$ *then* $x = y$.

Based on this property, we can restate case (4) of Lemma 3.1 as follows:

PROPOSITION 3.2. *A regular expression* r^+ *is deterministic if and only if* r *is deterministic and* $\mathcal{P}(r) = \mathsf{true}$.

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

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

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

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

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

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

Each rule in $\mathcal{D}$ corresponds to one case in Lemma 3.1. 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) in Lemma 3.1. 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 3.1. On the other hand, given a DREG r, we can construct a derivation tree for r, 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 3.1. *A regular expression* r *is deterministic if and only if* 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, 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 3.2. *Given a DREG* r, *if* r_1 *is a subexpression of* r, *and* r_2 *is a DREG with the same values of* First, followLast, λ *and* $\mathcal{P}$ *as* r_1, *then replacing* r_1 *by* r_2 *in* r *results in another DREG.*

Proof. Actually, we prove a stronger statement: replacing r_1 by r_2 in r results in a DREG r' with the same values of First, followLast, λ and $\mathcal{P}$ as r. If this statement holds, then the lemma follows immediately. We prove the statement by induction on the structure of r.

$r = a$, for $a \in \Sigma$: Then r_1 is r and the resulting expression r' is r_2. From the premise, namely, r_2 is a DREG with the same values of First, followLast, λ and $\mathcal{P}$ as r_1, the result follows.

$r = r_3 \mid r_4$: Here we only show the case where r_1 is a subexpression of r_3; the other case can be proved similarly. As r_1 is a subexpression of r_3, by induction on r_3, we know that replacing r_1 by r_2 in r_3 results in a DREG, denoted by r_3', which has the same values of First, followLast, λ and $\mathcal{P}$ as r_3. According to

Lemma 3.1, and the computations of First, followLast, λ and $\mathcal{P}$ (see Section 2 and Section 3.1), we know that $r_3' \mid r_4$ is also deterministic, and both $r_3' \mid r_4$ and $r_3 \mid r_4$ share the same values of First, followLast, λ and $\mathcal{P}$. Then the result follows.

The other cases can be proved similarly. □

Assume the alphabet of DREGs is $\Sigma = \{a_1, \ldots, a_n\}$. To construct the grammar for DREGs on Σ, we first define a finite set $\mathbb{X}$ of nonterminal symbols 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 of DREGs, denoted by $L(X^{S,R,\alpha,\beta})$, whose First, followLast, λ and $\mathcal{P}$ values are S, R, α and β, respectively. Note that for any regular expression r, any value of First, followLast, λ or $\mathcal{P}$ cannot be determined by the other values. For example, $(a_1|a_2)^+$ and $a_1 \cdot a_1^+|a_2^+$ share the same First, followLast, λ, but have different $\mathcal{P}$ values.

Having this property, we can construct a context-free grammar for DREGs (given in Figure 1). By Lemma 3.2, we can replace any subexpression r_1 in a DREG r by any DREG with the same First, followLast, λ and $\mathcal{P}$ values. Actually, 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: replace DREGs by nonterminals. Moreover, from the definition of $X^{S,R,\alpha,\beta}$, we need to consider the computations of First, 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 rules in $\mathcal{D}$; (2) each class of productions should conform to the computations of First, followLast, λ and $\mathcal{P}$; (3) nonterminals are selected carefully such that the related conditions in the rules of $\mathcal{D}$ hold.

Figure 1 shows the details of the grammar (denoted by G_d) we construct, where $\bigcup$ denotes a set of rules with the same left hand nonterminal, the subscript equations of $\bigcup$ are called as the side conditions of the corresponding rules, and the symbols (,), |, ·, + and ?, the alphabet symbols $a_i \in \Sigma$, and the empty symbol ε are all 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 case (G-Union). The productions for this case can be constructed in two steps: (1) compute the possible values of First, followLast, λ and $\mathcal{P}$ that the two operands of the union | can have; and (2) check the conditions to ensure determinism in $\mathcal{D}$.

Assume that the DREG r we want to generate satisfies: $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\lambda(r) = \alpha$ and $\mathcal{P}(r) = \beta$, that is, $X^{S,R,\alpha,\beta} \underset{G_d}{\overset{*}{\Longrightarrow}} r$. Because r is generated by the case (G-Union), clearly r should be of the form $r_1 \mid r_2$. First, consider the possible values $S_i, R_i, \alpha_i, \beta_i$ for these two subexpressions r_i such that $X^{S_i,R_i,\alpha_i,\beta_i} \underset{G_d}{\overset{*}{\Longrightarrow}} r_i$, where $i = 1, 2$. According to the computations of First, 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) $\beta = \mathcal{P}(r) = \mathcal{P}(r_1) \wedge \mathcal{P}(r_2) \wedge (\mathsf{followLast}(r_2) \cap \mathsf{First}(r_1) = \emptyset) \wedge (\mathsf{followLast}(r_1) \cap \mathsf{First}(r_2) = \emptyset) = \beta_1 \wedge \beta_2 \wedge (R_2 \cap S_1 = \emptyset) \wedge (R_1 \cap S_2 = \emptyset)$.

Moreover, to ensure determinism, the condition of the rule (Union) in $\mathcal{D}$, that is, (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 $r_1 \mid r_2$ must satisfy Conditions (1)-(5), which form the definition of Φ. The argumentation for DREGs of the form $r_1 \cdot r_2$ and the function Θ are similar.

Clearly, the grammars are context-free. Next, we study the correctness of these grammars: r is a DREG if and only if 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 3.3 and Lemma 3.4.

Lemma 3.3. *If $X^{S,R,\alpha,\beta} \overset{*}{\Rightarrow} r$, 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 $X^{S,R,\alpha,\beta} \overset{*}{\Rightarrow} r$. For convenience, we write X, X_1 and X_2 short for $X^{S,R,\alpha,\beta}$, $X^{S_1,R_1,\alpha_1,\beta_1}$ and $X^{S_2,R_2,\alpha_2,\beta_2}$, respectively.

Base case: If the derivation is one-step, then r is either ε or a_i, where $a_i \in \Sigma$. It is easy to see that the statement holds.

Inductive step: Suppose that the derivation takes $k+1$ steps ($k \geq 1$), and the statement holds for any derivation of no more than k steps.

Assume the rule used in the first step of the derivation is (G-Union). That is, the derivation is in the form $X \Rightarrow X_1 \mid X_2 \overset{*}{\Rightarrow} r$. In this case, r should be in form of $r_1 \mid r_2$, and we have that $X_i \overset{*}{\Rightarrow} r_i$ with steps no more than k, where $i = 1, 2$. By the inductive hypothesis on r_i, we know that r_i is a DREG with $\mathsf{First}(r_i) = S_i$, $\mathsf{followLast}(r_i) = R_i$, $\lambda(r_i) = \alpha_i$ and $\mathcal{P}(r_i) = \beta_i$. From the definition of the rule (G-Union), we have that $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$, $R_1 \cup R_2 = R$, $\alpha_1 \vee \alpha_2 = \alpha$ and $(\beta_1 \wedge \beta_2 \wedge (S_1 \cap R_2 = \emptyset) \wedge (R_1 \cap S_2 = \emptyset)) = \beta$. Because r_1 and 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 3.1. Moreover, from the computations of First, followLast, λ and $\mathcal{P}$, we can get $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\lambda(r) = \alpha$ and

$$\text{G-Base:} \quad X^{\{a_i\},\emptyset,\mathsf{false},\mathsf{true}} \to a_i \qquad X^{\emptyset,\emptyset,\mathsf{true},\mathsf{true}} \to \varepsilon$$

$$\text{G-Union:} \quad X^{S,R,\alpha,\beta} \to \bigcup_{\Phi(S,R,\alpha,\beta,S_1,R_1,\alpha_1,\beta_1,S_2,R_2,\alpha_2,\beta_2)} (X^{S_1,R_1,\alpha_1,\beta_1} \mid X^{S_2,R_2,\alpha_2,\beta_2})$$

$$\text{G-Concat:} \quad X^{S,R,\alpha,\beta} \to \bigcup_{\Theta(S,R,\alpha,\beta,S_1,R_1,\alpha_1,\beta_1,S_2,R_2,\alpha_2,\beta_2)} (X^{S_1,R_1,\alpha_1,\beta_1} \cdot X^{S_2,R_2,\alpha_2,\beta_2})$$

$$\text{G-Plus:} \quad X^{S,R,\alpha,\mathsf{true}} \to \bigcup_{R_1 \cup S = R} X^{S,R_1,\alpha,\mathsf{true}} +$$

$$\text{G-Que:} \quad X^{S,R,\mathsf{true},\beta} \to \bigcup_{\alpha_1 \in \{\mathsf{true},\mathsf{false}\}} X^{S,R,\alpha_1,\beta} ?$$

where $i = 1, \ldots, n$, the nonterminal symbol $X^{S,R,\alpha,\beta}$ is intended to describe the set of DREGs whose $\mathsf{First}, \mathsf{followLast}, \lambda$ and $\mathcal{P}$ values are respectively S, R, α and β, and the Boolean functions Φ and Θ are defined as follows:

$$\Phi(S,R,\alpha,\beta,S_1,R_1,\alpha_1,\beta_1,S_2,R_2,\alpha_2,\beta_2) \stackrel{def}{=} \mathsf{and} \begin{cases} (S_1 \cup S_2 = S) \wedge (S_1 \cap S_2 = \emptyset) \\ R_1 \cup R_2 = R \\ \alpha_1 \vee \alpha_2 = \alpha \\ \beta_1 \wedge \beta_2 \wedge (S_1 \cap R_2 = \emptyset) \wedge (R_1 \cap S_2 = \emptyset) = \beta \end{cases}$$

$$\Theta(S,R,\alpha,\beta,S_1,R_1,\alpha_1,\beta_1,S_2,R_2,\alpha_2,\beta_2) \stackrel{def}{=} \mathsf{and} \begin{cases} R_1 \cap S_2 = \emptyset \\ \alpha_1 \wedge \alpha_2 = \alpha \\ \alpha_1 \wedge (S_1 \cup S_2 = S) \wedge (S_1 \cap S_2 = \emptyset) \vee \neg\alpha_1 \wedge (S_1 = S) \\ \alpha_2 \wedge (R_1 \cup S_2 \cup R_2 = R) \vee \neg\alpha_2 \wedge (R_2 = R) \\ \theta(S_1,R_1,\alpha_1,\beta_1,S_2,R_2,\alpha_2,\beta_2) = \beta \end{cases}$$

$$\theta(S_1,R_1,\alpha_1,\beta_1,S_2,R_2,\alpha_2,\beta_2) \stackrel{def}{=} \mathsf{or} \begin{cases} \neg\alpha_1 \wedge \neg\alpha_2 \wedge (S_1 \cap R_2 = \emptyset) \\ \alpha_1 \wedge \neg\alpha_2 \wedge \beta_2 \wedge (S_1 \cap R_2 = \emptyset) \\ \neg\alpha_1 \wedge \alpha_2 \wedge \beta_1 \wedge (S_1 \cap R_2 = \emptyset) \wedge (S_1 \cap S_2 = \emptyset) \\ \alpha_1 \wedge \alpha_2 \wedge \beta_1 \wedge \beta_2 \wedge (S_1 \cap R_2 = \emptyset) \end{cases}$$

FIGURE 1. Context-free grammars for DREGs

$\mathcal{P}(r) = \beta$. Hence the result follows.

Similarly for the other rules used in the first step of the derivation. □

LEMMA 3.4. *If r is a DREG, then we have $X^{\mathsf{First}(r),\mathsf{followLast}(r),\lambda(r),\mathcal{P}(r)} \stackrel{*}{\Rightarrow} r$.*

Proof. We prove it by induction on the structure of r. For convenience, we write X_r short for the nonterminal $X^{\mathsf{First}(r),\mathsf{followLast}(r),\lambda(r),\mathcal{P}(r)}$.

The cases for $r = \varepsilon$ or $r = a$ are obvious, where $a \in \Sigma$.

Assume that r is in the form of $r_1 \mid r_2$. According to Lemma 3.1, we have that both r_1 and r_2 are also deterministic. By the inductive hypothesis, both r_1 and r_2 are derivable from G_d, that is, $X_{r_i} \stackrel{*}{\Rightarrow} r_i$ with $i = 1, 2$. From the computations of First, $\mathsf{followLast}$, λ and $\mathcal{P}$, and the fact that r is a DREG, we know that (1) $\mathsf{First}(r) = \mathsf{First}(r_1) \cup \mathsf{First}(r_2)$; (2) $\mathsf{First}(r_1) \cap \mathsf{First}(r_2) = \emptyset$; (3) $\mathsf{followLast}(r) = \mathsf{followLast}(r_1) \cup \mathsf{followLast}(r_2)$; (4) $\lambda(r) = \lambda(r_1) \vee \lambda(r_2)$; (5) $(\mathcal{P}(r_1) \wedge \mathcal{P}(r_2) \wedge (\mathsf{First}(r_1) \cap \mathsf{followLast}(r_2) = \emptyset) \wedge (\mathsf{followLast}(r_1) \cap \mathsf{First}(r_2) = \emptyset)) = \mathcal{P}(r)$. Therefore, X_r, X_{r_1} and X_{r_2} satisfy the conditions of (G-Union). According to the definition of G_d, we have that $X_r \Rightarrow X_{r_1} \mid X_{r_2} \stackrel{*}{\Rightarrow} r_1 \mid r_2$. That is, $X_r \stackrel{*}{\Rightarrow} r$. Hence, the result follows.

Similarly for the other cases. □

Based on the lemmas above, we can conclude that

THEOREM 3.2. *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, which follows from Lemma 3.3; and (2) All DREGs can be generated by G_d, which follows from Lemma 3.4. □

Let us discuss the size of the grammars. Assume the alphabet of DREGs is Σ. Firstly, we can compute the number of possible nonterminals easily: there are $2^{|\Sigma|}$ different possible First and $\mathsf{followLast}$ sets, and thus there are $2^{|\Sigma|} \cdot 2^{|\Sigma|} \cdot 2 \cdot 2 = 2^{2|\Sigma|+2}$ different possible nonterminals. Secondly, we compute the numbers of the productions of the same nonterminal $X^{S,R,\alpha,\beta}$ by case analysis. Clearly, there is only one production of $X^{S,R,\alpha,\beta}$ for (G-Base) when the side condition (*e.g.*, $S = \{a_i\}$, $R = \emptyset$, $\alpha = \mathsf{false}$ and $\beta = \mathsf{true}$) is satisfied. For (G-Plus) (resp. (G-Que)), there are $2^{|S|}$ (resp. 2) productions, when $S \subseteq R$ and $\beta = \mathsf{false}$ (resp. $\alpha = \mathsf{true}$). While for (G-Union) (resp. (G-Concat)), the number of possible productions of $X^{S,R,\alpha,\beta}$ is exactly the number of solutions to the Boolean function Φ (resp. Θ) for the given values S, R, α, β. Here we only show the case for Φ where both α and β are true; the other cases are similar. In this case, we have that there are 3 candidates for α_is and 1 candidate for β_is, and both $S_1 \cap R_2 = \emptyset$ and $S_2 \cap R_1 = \emptyset$ hold. Suppose that S_1 consists of i elements from $S \setminus R$ and j elements from $S \cap R$. Then S_2 is $S \setminus S_1$ and there are $\binom{|S \setminus R|}{i} \cdot \binom{|S \cap R|}{j}$ possible candidates for S_1. As $S_1 \cap R_2 = \emptyset$ (resp. $S_2 \cap R_1 = \emptyset$), the elements in $S_1 \cap R$ (resp. $S_2 \cap R$) should belong to R_1 (resp. R_2). Consider the elements

TABLE 1. Number of productions in G_d by cases

Case	Class	Condition	Production Number of $X^{S,R,\alpha,\beta}$	Total Number
1	(G-Base)	$S=\emptyset$, $R=\emptyset$, $\alpha=\mathsf{true}$, $\beta=\mathsf{true}$	$O(1)$	$O(1)$
2	(G-Base)	$S=\{a_i\}$, $R=\emptyset$, $\alpha=\mathsf{false}$, $\beta=\mathsf{true}$	$O(1)$	$O(\lvert\Sigma\rvert)$
3	(G-Union)	$\alpha=\mathsf{true}$, $\beta=\mathsf{true}$	$O(2^{\lvert S\rvert}\cdot 3^{\lvert R\setminus S\rvert})$	$O(8^{\lvert\Sigma\rvert})$
4	(G-Union)	$\alpha=\mathsf{true}$, $\beta=\mathsf{false}$	$O(2^{\lvert S\setminus R\rvert}\cdot 3^{\lvert R\setminus S\rvert}\cdot 6^{\lvert S\cap R\rvert})$	$O(12^{\lvert\Sigma\rvert})$
5	(G-Union)	$\alpha=\mathsf{false}$, $\beta=\mathsf{true}$	$O(2^{\lvert S\rvert}\cdot 3^{\lvert R\setminus S\rvert})$	$O(8^{\lvert\Sigma\rvert})$
6	(G-Union)	$\alpha=\mathsf{false}$, $\beta=\mathsf{false}$	$O(2^{\lvert S\setminus R\rvert}\cdot 3^{\lvert R\setminus S\rvert}\cdot 6^{\lvert S\cap R\rvert})$	$O(12^{\lvert\Sigma\rvert})$
7	(G-Concat)	$\alpha=\mathsf{true}$, $\beta=\mathsf{true}$	$O(3^{\lvert R\rvert})$	$O(8^{\lvert\Sigma\rvert})$
8	(G-Concat)	$\alpha=\mathsf{true}$, $\beta=\mathsf{false}$	$O(3^{\lvert R\setminus S\rvert}\cdot 4^{\lvert S\cap R\rvert})$	$O(9^{\lvert\Sigma\rvert})$
9	(G-Concat)	$\alpha=\mathsf{false}$, $\beta=\mathsf{true}$	$\max(O(5^{\lvert R\setminus S\rvert}), O(3^{\lvert\Sigma\rvert}))$	$O(12^{\lvert\Sigma\rvert})$
10	(G-Concat)	$\alpha=\mathsf{false}$, $\beta=\mathsf{false}$	$\max(O(5^{\lvert R\setminus S\rvert}\cdot 5^{\lvert S\cap R\rvert}), O(3^{\lvert\Sigma\rvert}))$	$O(12^{\lvert\Sigma\rvert})$
11	(G-Plus)	$\beta=\mathsf{true}$, $S\subseteq R$	$O(2^{\lvert S\rvert})$	$O(4^{\lvert\Sigma\rvert})$
12	(G-Que)	$\alpha=\mathsf{true}$	$O(1)$	$O(4^{\lvert\Sigma\rvert})$

in $R\setminus S$. Assume that R_1 contains k elements from it, which has $\binom{|R\setminus S|}{k}$ possible candidates. Then to satisfy the condition $R_1\cup R_2=R$, the other elements in $R\setminus S$ should belong to R_2. Moreover, R_2 can share some elements with R_1, which can only come from these k elements. So there are $\binom{k}{m}$ possible candidates for $R_1\cap R_2$, where m is the number of the common elements. Therefore, we can conclude that in the case (G-Base) there are at most $\sum_{i=0}^{|S\setminus R|}\sum_{j=0}^{|S\cap R|}\sum_{k=0}^{|R\setminus S|}\sum_{m=0}^{k}\binom{|S\setminus R|}{i}\cdot\binom{|S\cap R|}{j}\cdot\binom{|R\setminus S|}{k}\cdot\binom{k}{m}\cdot 3 = 3\cdot 2^{|S|}\cdot 3^{|R\setminus S|}$ productions, that is, in $O(2^{|S|}\cdot 3^{|R\setminus S|})$. Table 1 gives the number of possible productions for all cases.

Finally, we compute the total number of the possible productions in G_d by case analysis, which is also shown in Table 1. Likewise, here we only show the case for (G-Union) where both α and β are true. Both the sizes of S and R can range from 0 to $|\Sigma|$, and they may share some common elements. We consider $S\setminus R$, $R\setminus S$ and $S\cap R$ instead, which are pairwise disjoint. By combinatorics, there are $\binom{|\Sigma|}{|S\setminus R|}$, and $\binom{|\Sigma|-|S\setminus R|}{|S\cap R|}$ and $\binom{|\Sigma|-|S\setminus R|-|S\cap R|}{|R\setminus S|}$ possible candidates for $S\setminus R$, $R\setminus S$ and $S\cap R$, respectively. So the total production number in this case is $\sum_{|S\setminus R|=0}^{|\Sigma|}\sum_{|S\cap R|=0}^{|\Sigma|-|S\setminus R|}\sum_{|R\setminus S|=0}^{|\Sigma|-|S\setminus R|-|S\cap R|}\binom{|\Sigma|}{|S\setminus R|}\cdot\binom{|\Sigma|-|S\setminus R|}{|S\cap R|}\cdot\binom{|\Sigma|-|S\setminus R|-|S\cap R|}{|R\setminus S|}\cdot 3\cdot 2^{|S|}\cdot 3^{|R\setminus S|} = 3\cdot 8^{|\Sigma|}$, that is, in $O(8^{|\Sigma|})$. From the table, we can see that the size of G_d is in $O(12^{|\Sigma|})$.

It is known that the syntax of regular expressions is context-free. But as exemplified in Section 1, not all the subsets of a context-free language are context-free. Nevertheless, the set of DREGs preserves the well-nested parentheses, so we conclude that DREGs cannot be defined by regular grammars as well.

Proposition 3.3. *DREGs cannot be defined by regular grammars.*

Proof. Regular languages are closed under homomorphisms, and we can define an homomorphism mapping the set of DREGs to the Dyck language with one kind of parenthesis by projecting out symbols which are not parentheses. So we know that the set of DREGs cannot be regular. □

4. VALID PRODUCTIONS

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

For a general nonterminal $X^{S,R,\alpha,\beta}$, we call $X^{S,R,\alpha,\beta}$ is useful if there exists a DREG r such that $X^{S,R,\alpha,\beta}\overset{*}{\Rightarrow} r$. Obviously, if $S=\emptyset$, then only the empty string ε satisfies this condition, and thus we have $R=\emptyset$, $\alpha=\mathsf{true}$, and $\beta=\mathsf{true}$. Moreover, when $S\cap R=\emptyset$, by Proposition 3.1, we have that $\beta=\mathsf{true}$. These two conditions are clearly necessary for $X^{S,R,\alpha,\beta}$ being useful. We show that these two conditions are also sufficient.

Lemma 4.1. *Given S, R, α, and β, the nonterminal $X^{S,R,\alpha,\beta}$ is useful if and only if the following two conditions hold:*

(1) $(S=\emptyset)\rightarrow((R=\emptyset)\wedge(\alpha=true)\wedge(\beta=true))$;
(2) $(S\cap R=\emptyset)\rightarrow(\beta=true)$.

Proof. ($\Leftarrow$) We will show that if the conditions hold, then we can construct a DREG r such that $\mathsf{First}(r)=S$, $\mathsf{followLast}(r)=R$, $\lambda(r)=\alpha$ and $\mathcal{P}(r)=\beta$. According to Lemma 3.4, we know that $r\in L(X^{S,R,\alpha,\beta})$, that is, $X^{S,R,\alpha,\beta}$ is useful. We prove it by case analysis on the conditions.

Assume that $S\cap R\neq\emptyset$ and $S\neq\emptyset$. In this case, both Condition (1) and (2) hold trivially. Without loss of generality, we assume $S=\{a_1,\ldots,a_n,b_1,\ldots,b_{m_1}\}$ and $R=\{a_1,\ldots,a_n,c_1,\ldots,c_{m_2}\}$, and thus $S\cap R=\{a_1,\ldots,a_n\}$, where $a_i,b_j,c_k\in\Sigma$ are different symbols, $1\leq n$, $0\leq m_1$ and $0\leq m_2$. The DREG r is constructed as follows:

- If $\alpha=\mathsf{true}$ and $\beta=\mathsf{true}$, then we construct the expression $r=\varepsilon\mid(a_1^+\mid\ldots\mid a_n^+\mid b_1\mid\ldots\mid b_{m_1})(\varepsilon|c_1\mid\ldots\mid c_{m_2})$.

- If $\alpha = \mathsf{true}$ and $\beta = \mathsf{false}$, then we construct the expression $r = \varepsilon \mid (a_1 \mid \ldots \mid a_n \mid b_1 \mid \ldots \mid b_{m_1})(\varepsilon \mid a_1 \mid \ldots \mid a_n \mid c_1 \mid \ldots \mid c_{m_2})$.
- If $\alpha = \mathsf{false}$ and $\beta = \mathsf{true}$, then we construct the expression $r = (a_1^+ \mid \ldots \mid a_n^+ \mid b_1 \mid \ldots \mid b_{m_1})(\varepsilon \mid c_1 \mid \ldots \mid c_{m_2})$.
- If $\alpha = \mathsf{false}$ and $\beta = \mathsf{false}$, then we construct the expression $r = (a_1 \mid \ldots \mid a_n \mid b_1 \mid \ldots \mid b_{m_1})(\varepsilon \mid a_1 \mid \ldots \mid a_n \mid c_1 \mid \ldots \mid c_{m_2})$.

For each case above, we can verify that the constructed expression r is deterministic, and satisfies that $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\lambda(r) = \alpha$ and $\mathcal{P}(r) = \beta$.

Assume $S \cap R = \emptyset$ and $S \neq \emptyset$. In this case, Condition (1) holds trivially, and by Condtion (2), we have $\beta = \mathsf{true}$. Let $S = \{b_1, \ldots, b_{m_1}\}$ and $R = \{c_1, \ldots, c_{m_2}\}$, where the symbols are the same as above, except that $1 \leq m_1$, since S is not empty. Similarly, we construct the DREG r as follows:

- If $\alpha = \mathsf{true}$, then we construct the expression $r = \varepsilon \mid (b_1 \mid \ldots \mid b_{m_1})(\varepsilon \mid c_1 \mid \ldots \mid c_{m_2})$.
- If $\alpha = \mathsf{false}$, then we construct the expression $r = (b_1 \mid \ldots \mid b_{m_1})(\varepsilon \mid c_1 \mid \ldots \mid c_{m_2})$.

Similarly, we can verify that $r \in L(X^{S,R,\alpha,\beta})$ for each case above.

Assume $S = \emptyset$ and thus $S \cap R = \emptyset$ holds trivially. In this case, we have that $R = \emptyset$, $\alpha = \mathsf{true}$, and $\beta = \mathsf{true}$ from these two conditions. Clearly, the expression $r = \varepsilon$ satisfies that $\mathsf{First}(r) = \emptyset$, $\mathsf{followLast}(r) = \emptyset$, $\lambda(r) = \mathsf{true}$ and $\mathcal{P}(r) = \mathsf{true}$.

($\Rightarrow$) Let us consider a general useful nonterminal $X^{S,R,\alpha,\beta}$. If $S \neq \emptyset$ and $S \cap R \neq \emptyset$, then both conditions hold trivially. In the following, we consider the nontrivial cases.

Assume that $S = \emptyset$. As $X^{S,R,\alpha,\beta}$ is useful, there exists a DREG r such that $r \in L(X^{S,R,\alpha,\beta})$. According to Lemma 3.3, we have $\mathsf{First}(r) = S = \emptyset$. Without loss of generality, we assume that regular expressions are not $\emptyset$. Thus, there exists only one expression ε satisfying the condition $\mathsf{First}(r) = \emptyset$. Accordingly, $\mathsf{followLast}(r) = \emptyset$, $\lambda(r) = \mathsf{true}$ and $\mathcal{P}(r) = \mathsf{true}$. By Lemma 3.3 again, we get $R = \emptyset$, $\alpha = \mathsf{true}$, and $\beta = \mathsf{true}$, and thus both conditions hold.

Assume that $S \neq \emptyset$ and $S \cap R = \emptyset$. In this case, Condition (1) holds trivially. So we only need to consider Condition (2), that is, to prove $\beta = \mathsf{true}$. As $X^{S,R,\alpha,\beta}$ is useful, there exists a DREG r such that $r \in L(X^{S,R,\alpha,\beta})$. By Lemma 3.3, we have that $\mathsf{First}(r) = S$, $\mathsf{followLast}(r) = R$, $\lambda(r) = \alpha$ and $\mathcal{P}(r) = \beta$. From the premise $S \cap R = \emptyset$, we get $\mathsf{First}(r) \cap \mathsf{followLast}(r) = \emptyset$, and so does $\mathsf{First}(r^\sharp) \cap \mathsf{followLast}(r^\sharp) = \emptyset$. Therefore, the following condition holds: $\forall x \in \mathsf{followLast}(r^\sharp), \forall y \in \mathsf{First}(r^\sharp)$, if $x^\sharp = y^\sharp$ then $x = y$. By Proposition 3.1, we have that $\mathcal{P}(r) = \mathsf{true}$, that is, $\beta = \mathsf{true}$. □

We use G_o to denote such optimized grammars with only valid productions. According to Lemma 4.1, we know that G_o denotes the same language as G_d.

In the following, we discuss the effectiveness of Lemma 4.1, that is, we show how many nonterminals and productions are approximately ruled out by this lemma. If $S = \emptyset$, then there are $2^{|\Sigma|} \cdot 2 \cdot 2 = 2^{|\Sigma|+2}$ different nonterminals. But only one among them is useful. Therefore, $2^{|\Sigma|+2} - 1$ nonterminals are ruled out by Condition (1). Next, we consider the useless nonterminals such that $|S| \neq \emptyset$, $S \cap R = \emptyset$ and $\beta = \mathsf{false}$, which are ruled out by Condition (2). Assuming $|S| = i$, there are $\binom{|\Sigma|}{i}$ possible sets for S. As $S \cap R = \emptyset$, R must be a subset of $\Sigma \setminus S$, which has $2^{|\Sigma|-i}$ cases. So there are at most $\binom{|\Sigma|}{i} \cdot 2^{|\Sigma|-i} \cdot 2$ useless nonterminals for $|S| = i$, and thus the number of nonterminals ruled out by Condition (2) is $\sum_{i=1}^{|\Sigma|}(\binom{|\Sigma|}{i} \cdot 2^{|\Sigma|-i} \cdot 2) = 2 \cdot (3^{|\Sigma|} - 2^{|\Sigma|})$. In total, the number of nonterminals ruled out by Lemma 4.1 is $2 \cdot 3^{|\Sigma|} + 2^{|\Sigma|+1} - 1$, which is in $O(3^{|\Sigma|})$.

Let us consider the number of possible invalid productions. There are two reasons for a production to be invalid: (1) the nonterminal on the left-hand side is useless, and (2) one of the nonterminals on the right-hand side is useless. Clearly, if a nonterminal $X^{S,R,\alpha,\beta}$ is useless, then all its productions listed in Figure 1 are invalid. Let us consider the case for (G-Union) where both α and β are true again. The situation to make $X^{S,R,\alpha,\beta}$ useless is that $S = \emptyset$ but $R \neq \emptyset$. So there are $\sum_{|R \setminus S|=1}^{|\Sigma|} \binom{|\Sigma|}{|R \setminus S|} \cdot 3 \cdot 2^{|S|} \cdot 3^{|R \setminus S|} = 3 \cdot (4^{|\Sigma|} - 1)$, that is, in $O(4^{|\Sigma|})$. The other cases can be done similarly.

Assume that $X^{S,R,\alpha,\beta}$ is useful and the first nonterminal $X^{S_1,R_1,\alpha_1,\beta_1}$ on the right-hand side is useless. Take the case for (G-Union) where $\alpha = \mathsf{true}$, $\beta = \mathsf{false}$, and both $S_1 \cap R_2 = \emptyset$ and $S_2 \cap R_1 = \emptyset$ hold as an example. In this case, we have $\beta = \beta_1 \wedge \beta_2$. Assume that $X^{S_1,R_1,\alpha_1,\beta_1}$ is useless due to $S_1 \cap R_1 = \emptyset$ and $\beta_1 = \mathsf{false}$. This case is quite similar to the one for (G-Union) where both α and β are true. According to the analysis in Section 3.2, we have $|S_1 \cap R_1| = j$. To satisfy $S_1 \cap R_1 = \emptyset$, j should be 0. Therefore, in this case there are $\sum_{i=0}^{|S \setminus R|} \sum_{j=0}^{0} \sum_{k=0}^{|R \setminus S|} \sum_{m=0}^{k} \binom{|S \setminus R|}{i} \cdot \binom{|S \cap R|}{j} \cdot \binom{|R \setminus S|}{k} \cdot \binom{k}{m} \cdot 6 = 6 \cdot 2^{|S \setminus R|} \cdot 3^{|R \setminus S|}$ invalid productions. Moreover, as $X^{S,R,\alpha,\beta}$ is useful and $\beta = \mathsf{false}$, $S \cap R$ (and S) should not be empty. Hence, in the grammar there are $\sum_{|S \cap R|=1}^{|\Sigma|} \sum_{|S \setminus R|=0}^{|\Sigma|-|S \cap R|} \sum_{|R \setminus S|=0}^{|\Sigma|-|S \cap R|-|S \setminus R|} \binom{|\Sigma|}{|S \cap R|} \cdot \binom{|\Sigma|-|S \cap R|}{|S \setminus R|} \cdot \binom{|\Sigma|-|S \cap R|-|S \setminus R|}{|R \setminus S|} \cdot 6 \cdot 2^{|S \setminus R|} \cdot 3^{|R \setminus S|} = 6(7^{|\Sigma|} - 6^{|\Sigma|})$ invalid productions in total, that is, in $O(7^{|\Sigma|})$.

Table 2 lists some numbers of invalid productions by cases, where the first column denotes the cases shown in Table 1. The computation of the invalid production numbers due to that $X^{S_2,R_2,\alpha_2,\beta_2}$ is useless is similar to the one of $X^{S_1,R_1,\alpha_1,\beta_1}$ and thus is not given. From this table, we can see that the number of productions ruled out by Lemma 4.1 is at least in $O(10^{|\Sigma|})$, which indicates that this lemma is effective to reduce the size of grammars. Nevertheless, the size of G_o is still in $O(12^{|\Sigma|})$.

TABLE 2. Numbers of invalid productions by cases

Case	$X^{S,R,\alpha,\beta}$ is useless	$X^{S_1,R_1,\alpha_1,\beta_1}$ is useless
3	$O(4^{\lvert\Sigma\rvert})$	$O(6^{\lvert\Sigma\rvert})$
4	$O(6^{\lvert\Sigma\rvert})$	$O(10^{\lvert\Sigma\rvert})$
5	$O(4^{\lvert\Sigma\rvert})$	$O(6^{\lvert\Sigma\rvert})$
6	$O(6^{\lvert\Sigma\rvert})$	$O(10^{\lvert\Sigma\rvert})$
7	$O(4^{\lvert\Sigma\rvert})$	$O(6^{\lvert\Sigma\rvert})$
8	$O(5^{\lvert\Sigma\rvert})$	$O(8^{\lvert\Sigma\rvert})$
9	$O(6^{\lvert\Sigma\rvert})$	$O(6^{\lvert\Sigma\rvert})$
10	$O(7^{\lvert\Sigma\rvert})$	$O(10^{\lvert\Sigma\rvert})$
11	$O(2^{\lvert\Sigma\rvert})$	0
12	$O(3^{\lvert\Sigma\rvert})$	$O(1)$

TABLE 3. $|G_d|$ and $|G_o|$ over small alphabets

$\lvert\Sigma\rvert$	1	2	3	4	5
$\lvert V_d\rvert$	16	64	256	1024	4096
$\lvert V_o\rvert$	7	39	187	831	3547
$\lvert G_d\rvert$	208	3297	43698	542387	6553772
$\lvert G_o\rvert$	46	815	14904	240481	3520010

To show the effectiveness of this lemma, we list the numbers of nonterminals and productions in $|G_d|$ and $|G_o|$ over small alphabets in Table 3, where V_d (resp. V_o) denotes the nonterminals in G_d (resp. G_o). Note that permutation of symbols in First and followLast sets does not matter. From the table, we can see that (1) both the number of productions in $|G_d|$ and $|G_o|$ increases rapidly; (2) the numbers of productions in G_o are much fewer than the ones in G_d.

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 case (G-Base) is trivial, since these productions can generate expressions in one step. In the following, we show that there are valid productions in the other classes. Actually, the conditions in the productions correspond to the conditions for an expression to be deterministic and the processes of computing First, followLast, λ and $\mathcal{P}$. Let us consider the following DREGs: $r_1 = \varepsilon \mid a$, $r_2 = aa$, $r_3 = a^+$ and $r_4 = a?$ for any symbol $a \in \Sigma$, which correspond to the other classes of productions in G_d. Then it suffices to construct one valid production for each expression. The productions we constructed are as follows: $(P1)$ $X_2 \to X_0 \mid X_1$, $(P2)$ $X_1 \to X_1 \cdot X_1$, $(P3)$ $X_3 \to X_1+$, 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 all X_is are useful, all these productions are valid.

5. RANDOM GENERATION OF DREGS

We have shown in Section 3.2 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, whose sizes are as close to a predefined size as possible, as uniformly at random and efficiently as possible.

5.1. Random Generation Algorithm

With the grammars presented in Section 3.2, a naive idea is to first construct the whole grammar for a given alphabet, and then use the existing methods [28, 30, 32, 33, 38] to randomly generate DREGs from the constructed grammar. However, due to the facts that the grammars are too large to construct fully (see Section 3.2), and that existing methods require some pre-processing of the whole grammar, the existing methods become inefficient and thus are not suitable for the grammars of DREGs. In existing methods, pre-processing is used to compute some information to make the random generation efficient. Hence the key problem is how to make the random generation efficient without the construction and the pre-processing of the whole grammar. Thanks to the parameters in the nonterminals of the grammars for DREGs, we can take full advantage of such 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 4.1, we can directly construct the valid productions. Otherwise, we need to construct the whole grammar to check the validity of each production, which could lead to an inefficient generation.
- Select the productions of a nonterminal uniformly. Based on the parameters, we can compute the numbers of the valid productions of a nonterminal by case analysis, which enable us to select the productions as uniformly as possible.
- Impose a conservative size condition to make the generation efficient. More specifically, if the size of the DREG that is being generated exceeds a given size, the generation terminates with a failure. This is similar to most existing work, but different in that the size of the generated DREG is not larger than the given one, while existing work requires that the size[4] is exactly the given one. This weak condition enables us to avoid the pre-processing of the whole grammar, which is required by existing work.
- Employ a size control mechanism to improve the efficiency of generation. In detail, we overestimate the possible (minimum) size l_e for the current DREG that is being generated. Then we compare l_e with the given size l. If $l_e \geq l$, the generation

[4]Indeed, existing work uses the notion of *length*. But for consistency, we use *size* instead.

starts to select the productions with smaller First and followLast sets randomly, which we call *the smaller selection*. As a result, the selected First and followLast become smaller and smaller until the case (G-Base) of the grammar. Therefore, the generation will terminate eventually. Moreover, the smaller selection can make the generation more uniform.

Algorithm 1 Random Generation Algorithm

Input: the size l and the alphabet Σ
Output: a DREG s s.t. $|s| \leq l$ or failure

```
1: generate a useful nonterminal X_s randomly and put it into stack stack
2: set current expression s_c ← ε, size l_c ← 0
3: set size control off LC ← false
4: while stack is nonempty and l_c ≤ l do
5:    X ← pop stack
6:    if X is terminal then
7:       s_c ← s_c · X,  l_c ← l_c + 1 if X ∈ Σ
8:    else
9:       if LC is false then
10:         l_r ← estSize(stack),  LC ← (l_c + l_r > l)
11:      end if
12:      p = selPro(X, LC)
13:      if p is failure then
14:         return failure
15:      end if
16:      push rhs(p) into stack reversely
17:   end if
18: end while
19: if stack is empty then
20:    return s_c
21: else
22:    return failure
23: end if
```

The random generation algorithm is shown in Algorithm 1, which takes an alphabet Σ and a size l as input, and returns a DREG with size no larger than l, or a failure. The algorithm starts with a stack with a random useful nonterminal (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 two conditions in Lemma 4.1. And the algorithm initializes the current expression s_c as empty string ε and its current size l_c as 0 (Line 2), and sets the size control mechanism off (Line 3). Then the algorithm proceeds with each (top) symbol in the stack until either the stack is empty or the size of the current expression exceeds the required one ($l_c \geq l$) (Lines 4-18). In detail, if the processed symbol X is a terminal, then the algorithm concatenates it with the current expression, and increases the size by 1 if $X \in \Sigma$ (Lines 6-7). Otherwise, the algorithm uses the size function $estSize$ (see Section 5.2) to estimate a possible size l_r that the current stack can generate, and compares it with the remaining size (*i.e.*, the required size l minus the current one l_c) to trigger the size control mechanism (Line 10). Next, the algorithm tries to construct a production p for the symbol X following the rules in Figure 1 (Line 12) depending on the size control: If the size control mechanism is off, p is selected as uniformly at random as possible from all valid productions of X; otherwise, p is randomly selected from those productions such that each nonterminal symbol X' on the right-hand side has a smaller First set S and a smaller followLast set R than the corresponding ones in X (see Section 5.3 for more details). If the construction fails, the algorithm terminates with a failure (Lines $13-15$); otherwise, it pushes all the symbols on the right-hand side of p into the stack reversely (Line 16), and continues with the next possible symbol on the top of the stack. Finally, if the stack is empty, the algorithm returns the generated DREG (Lines $19-20$); otherwise, it returns a failure (Line 22).

5.2. Size Estimation

In this section we discuss how to define the size estimating function $estSize$ on symbols (Line 10 in Algorithm 1) to make the random generation efficient without any pre-processing of the whole grammar. Clearly, the size of a sentence generated from a terminal symbol is fixed (*i.e.*, 1), while it is variant when generated from a nonterminal symbol. So in the following, we focus on the definition of $estSize$ on nonterminals. To do that, we first review existing solutions, namely, the minimum size of the sentences generated from a nonterminal. But defining $estSize$ as the minimum size is problematic in our setting (see the discussion below). Then we show how to refine the minimum size definition with respect to the smaller selection, employed in our algorithm mentioned above.

Existing work on sentence generation with size control mechanism [42, 39] takes the the minimum size of the sentences generated from a nonterminal as a reference to guide the generation. Likewise, we can also define the size estimating function $estSize$ on a nonterminal as the minimum size of the DREGs generated from this nonterminal. The solutions in existing work require some pre-processing of the whole grammar to calculate the minimum size and the corresponding sentence that can be generated from a nonterminal. As we want to avoid pre-processing, the calculation is not applicable in our setting.

Nevertheless, thanks to the parameters (*i.e.*, the First set S and the followLast set R) in a nonterminal, we can figure out that any DREG s generated from the nonterminal $X^{S,R,\alpha,\beta}$ satisfies that the size $|s|$ of s is not less than the size of the union of its First set S and its followLast set R, since both sets are contained in the set $sym(s)$ of symbols occurring in s. This is formalized as the following lemma.

Lemma 5.1. *Given S, R, α, and β, then for each $s \in L(X^{S,R,\alpha,\beta})$, $|S \cup R| \leq |s|$.*

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

Lemma 5.1 gives us a low bound for the minimum size of generated DREGs. To check whether this low bound is exactly the minimum size or very close to it, we try to construct DREGs from different possible nonterminals such that their sizes are as close to $|S \cup R|$ as possible. Examples of such constructed DREGs are listed in Table 4, where $P_1 = \{a_1, \ldots, a_m\}$, $P_2 = \{c_1, \ldots, c_p\}$ and $P_3 = \{b_1, \ldots, b_n\}$, $a_i, b_j, c_k \in \Sigma$ are different symbols, $1 \leq m$, $1 \leq n$, and $1 \leq p$. From these constructed DREGs, we can see that, for a useful nonterminal $X^{S,R,\alpha,\beta}$, there exists a DREG with size $|S \cup R|$ if β is true, or with size $|S \cup R| + 1$ if β is false. Consequently, we declare that the low bound given by Lemma 5.1 is a reasonable solution to $estSize$[5].

As the calculation of the minimum size is solved, let us go back to the generation. When the current size l_c plus the estimated size l_r (*i.e.*, the possible size) exceeds the required size l, there is no more "space" for us to "expand" the DREG and we will trigger the size control mechanism. Therefore, we have to guide the generation to select the productions for the nonterminals in the stack such that the total size of the (sub-)DREGs generated from them is not larger than the remaining size (*i.e.*, the estimated size l_r). This is to say, we have to select the productions that leads to a DREG with the minimum size. As mentioned above, existing work [42, 39] performs some pre-proceeding on the whole grammar to record such productions (and the minimum sentences) for each nonterminal. While in our setting, as the whole grammar is "unknown", so are the productions leading to a minimum DREG. Moreover, even if the minimum DREGs were known (such as the DREGs presented in Table 4), the generation should not return them directly, since it would skip over the other candidates. In a word, defining $estSize$ as the minimum size could make the generation less uniform over the DREGs whose sizes are close to l.

As shown in Section 5.1, our solution employs a different size selection mechanism, to which we refer as *the smaller selection*. The smaller selection selects the productions randomly from the ones with smaller First sets and followLast sets, so it can select as more candidates as possible and lead the generation to terminating. Accordingly, we should refine $estSize$ with respect to the minimum size of generated DREGs under this smaller selection.

To satisfy the size condition $l_c + l_r \leq l$, the size generated by the smaller selection should be smaller than the estimated size l_r. This indicates that the estimated size l_r should be as large as possible. Hence, the earlier the size control mechanism is triggered on, the better it is. On the other hand, we would like to generate the DREGs whose sizes are as close to the given one as possible. This requires the generation to start the size control mechanism as late as possible. Based on the analysis above, we refine $estSize$ as *the maximum of the minimum sizes* for each class of productions under the smaller selection.

The remaining is to show how to calculate the maximum of the minimum sizes. For that, let us consider the production cases of a nonterminal $X^{S,R,\alpha,\beta}$. It is clear that the minimum size for (G-Base) is 1. For (G-Plus) and (G-Que), there is only one nonterminal $X^{S_1,R_1,\alpha_1,\beta_1}$ on the right-hand side, and we know $|S_1 \cup R_1| = |S \cup R|$ from the side conditions. So, by Lemma 5.1, the possible minimum sizes of $X^{S,R,\alpha,\beta}$ and $X^{S_1,R_1,\alpha_1,\beta_1}$ are almost the same. While for (G-Union) and (G-Concat), there are two nonterminals $X^{S_1,R_1,\alpha_1,\beta_1}$ and $X^{S_2,R_2,\alpha_2,\beta_2}$ on the right-hand side. Under the smaller selection, we also need to consider the following conditions for determinism: $S = S_1 \cup S_2$, $S_1 \cap S_2 = \emptyset$ and $R_1 \cup R_2 = R$. Then we know that the minimum size of $X^{S,R,\alpha,\beta}$ is no larger than the one of the sentential form on the right-hand side, since we have $|S \cup R| \leq |S_1 \cup R_1| + |S_2 \cup R_2|$. Hence, the possible minimum size increases, if (G-Union) and (G-Concat) are selected. From these, we can further deduce that $|S \cup R| \leq |S_1 \cup R_1| + |S_2 \cup R_2| \leq |S_1 \cup R| + |S_2 \cup R| \leq |S_1| + |R| + |S_2| + |R| = |S| + 2 * |R|$. That is to say, at most an additional $|R|$ can be added at a time. Moreover, these two cases can be selected continuously ($|S|$ times at most) until the case (G-Base) (*i.e.*, $|S_i| = 1$) is selected. So the possible minimum size can be $|S| + |R| + |S| * |R|$ at most. Therefore, we have the following lemma.

Lemma 5.2. *Let $X^{S,R,\alpha,\beta}$ be a nonterminal. Under the smaller selection, the maximum of the possible minimum sizes of DREGs generated from $X^{S,R,\alpha,\beta}$ is no larger than $|S \cup R| + |S| * |R|$.*

Proof. This can be proved by induction on the derivation trees of DREGs.
(G-Base): Trivial.

(G-Union): From the definition of Φ, we have that $S = S_1 \cup S_2$, $S_1 \cap S_2 = \emptyset$, and $R_1 \cup R_2 = R$. Let the minimum sizes of DREGs from $X^{S,R,\alpha,\beta}, X^{S_1,R_1,\alpha_1,\beta_1}, X^{S_2,R_2,\alpha_2,\beta_2}$ be m, m_1, m_2, respectively. Then we can bound m as follows.

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

(G-Concat): Due to the smaller selection, we have $|S_i| < |S|$ and $|R_i| < |R|$ $(i = 1, 2)$. It means that

[5]Similarly, we can give the low bounds with respect to *length* by case analysis. Different from the setting of *size*, the low bounds for the setting of *length* have a larger variance.

TABLE 4. DREGs with size $|S \cup R|$ and $|S \cup R| + 1$

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

we have to select the productions such that $\alpha_i =$ true[6], 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 (G-Union), we can prove that the minimum size is at most $|S \cup R| + |S| * |R|$.
(G-Plus): Let the minimum sizes of DREGs generated from $X^{S,R,\alpha,\text{true}}$ and $X^{S,R_1,\alpha,\text{true}}$ be 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 $R_1 \subseteq R$.

(G-Que): Similar to the case of (G-Plus). □

Actually, we also tried some other choices, such as the approximate average of the possible minimum sizes, *i.e.*, $(|S \cup R| + 0.5 * |S| * |R|)$. More details are presented in Section 6.

5.3. Production Selection

In this section, we first discuss the difficulty in the production selection to guide the generation to generate DREGs of the same size uniformly. Then we discuss how to select a production for a nonterminal $X^{S,R,\alpha,\beta}$ (*selPro* in Line 12 of Algorithm 1) as uniformly over the possible ones as possible, without any pre-processing of the whole grammar.

Theoretically, similar to Dong's algorithm [38], we can compute the following information by induction: (1) the number of DREGs with size l generated from a nonterminal X (*i.e.*, the *Hierarchy* information in [38]); (2) the number of DREGs with size l generated from a production P (*i.e.*, the *Cluster* information in [38]); (3) the number of DREGs generated from a production P satisfying that the size of sub-DREG generated from the ith nonterminal X_i on the right-hand side of P is l_i (*i.e.*, the *Cube* information in [38]). With these information, we can guide the generation to generate DREGs, uniformly distributed over the ones of the same size. However, due to the grammar is too large (see Section 3.2), the computational load is too large and it requires too much space to record these information. We failed in the grammars with alphabet size larger than 7, when we tried to generate DREGs in this way (see Section 6.1). An alternate way is to compute these information when it is needed, but the computation would be redundant and inefficient.

[6]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.

Algorithm 2 Production Selection Algorithm *selPro*

Input: a nonterminal $X^{S,R,\alpha,\beta}$ and size control status LC
Output: a production p or failure
1: **if** $X^{S,R,\alpha,\beta}$ is useless **then**
2: **return** failure
3: **end if**
4: **if** LC is false **then**
5: **for** $i = 1, \ldots, 12$ (the 12 cases in Table 1) **do**
6: $n_i =$ the production number of $X^{S,R,\alpha,\beta}$ for Case i
7: **end for**
8: **else**
9: **for** $i = 1, \ldots, 12$ **do**
10: $n_i =$ the production number of $X^{S,R,\alpha,\beta}$ s.t. $\forall X' \in rhs(p)$. $X'.S \subset X.S$ and $X'.R \subset X.R$ for Case i
11: **end for**
12: **end if**
13: $t = \sum_{i=1}^{12} n_i$
14: **if** $t = 0$ **then**
15: **return** failure
16: **end if**
17: generate a random number n in $\{1, \ldots, t\}$
18: let c s.t. $\sum_{i=1}^{c-1} n_i < n \leq \sum_{i=1}^{c} n_i$
19: generate $S_i, R_i, \alpha_i, \beta_i$ w.r.t. Case c and LC, yielding p
20: **return** p

As shown in Section 3.2 and Section 4, according to the parameters in the nonterminals, we can compute the (approximate) numbers of all the productions and of the invalid ones by case analysis, and thus the number of the valid productions. Based on these numbers, we can select the productions uniformly. It is similar for the smaller selection, plus the consideration of the additional conditions. Algorithm 2 shows the details of our production selection, which consists of two steps: (1) production case selection (Lines $4-18$) and (2) production parameter selection (Line 19). Firstly, according to Tables 1 and 2, we compute the numbers of valid productions by cases[7] (Lines $4-12$). Taking these numbers as a reference of probabilities, we select a case c randomly (Lines $13-18$). Secondly, following the conditions of Case c and LC, we randomly generate the parameters $S_i, R_i, \alpha_i, \beta_i$, and then construct the production p (Line 19).

For example, assume that the case for (G-Union) where both α and β are `true` is selected. According the discussion in Section 3.2, we will select the possible values for i, j, k and m such that the resulting production p is valid. Similar to case selection, we can compute the number of candidates for j, k and m for each possible value of i, and then select a value for i based on these numbers. The selections of j, k and m are proceeded in a similar way. It is easy to see that this selection enables us to select a production as uniformly as possible. While in practice, some cases may be too tedious to compute, so for efficiency and simplicity, we make a compromise for these cases: we select i, j, k and m uniformly among their possible values in proper order.

Without the size condition, the uniform selection of production will generate a DREG uniformly among all the DREGs, and thus is prone to generate a large DREG. So we think that our generation generates DREGs as close to a given size as possible when LC is off. When LC is on, thanks to the smaller selection and the definition of $estSize$, it tries to select the DREGs whose sizes are close to the given size as uniformly as possible. It is not easy to figure out the exact relation between the distribution of productions and the one of the corresponding DREGs for our generation, but our generation can generate DREGs efficiently and is viable in practice (see Section 6 for more details).

6. EXPERIMENTS

We have implemented a prototype of the generator to construct the DREG grammars and to randomly generate DREGs from these grammars in *Java*.

Using our prototype, we conducted a series of experiments to evaluate our solution. (1) Firstly, we conducted experiments to see whether the DREG grammars can be constructed and further be easily used in practice. (2) Secondly, we conducted experiments to test the performance of our random generation algorithm on different estimating functions. (3) Thirdly, to test the efficiency further, we also conducted experiments to compare our algorithm with the RE algorithm, which first generates regular expressions and then selects the deterministic ones. (4) 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.

[7] Other taxonomies are fine as well.

TABLE 5. Grammar construction for given alphabet Σ

$\lvert\Sigma\rvert$	$\lvert G_o\rvert$	$Time$	$Stored$	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

6.1. DREG Grammar Construction

Given an alphabet Σ, constructing the grammar is easy, just by constructing all the valid productions according to the rules presented in Figure 1. Thanks to Lemma 4.1, those invalid productions can be directly thrown away, without any additional check on the whole grammar. Table 5 shows the experimental results in 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 storing the grammar. Although the grammars can be constructed easily for small alphabets, the result shows that the time and space needed by the grammars are exponential in $|\Sigma|$. This is consistent with the number of productions given in Section 4, *i.e.*, in $O(12^{|\Sigma|})$.

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 [38], which is a linear time algorithm to enumerate sentences from a CFG and can be used as a sentence generator provided with a randomizer [39]. But we only succeed for the grammars whose alphabet sizes are smaller than 7, due to the large memory required by the maintenance of the whole grammar and the information required by the generation.

However, in some cases, only some productions for a specific non-terminal symbol are required, rather than the whole grammars. For example, when using these grammars for sentence generation (*e.g.*, our random generation) only one production is required for

each nonterminal symbol at a time. Moreover, the experimental result also shows that the average time for constructing a production is very low (almost in the order of $1us$). This enables us to efficiently generate DREGs by only constructing the productions when they are needed.

6.2. DREG Generation

In this section, we present some experiments to evaluate our random generation algorithm. First, as discussed in Section 5.2, the size estimating function $estSize$ affects the generation, so we use different size functions to conduct experiments on grammars with different alphabet sizes (ranging from 1 to 20). Second, experiments on grammars with large sizes (ranging from 21 to 27)[8] are conducted as well.

Experiments on different size functions. In these experiments, we respectively generate 100 DREGs with size no larger than 10, 20 and 50 from the grammars, whose alphabet size ranges from 1 to 20. For each generation, we use as $estSize$ the maximum of the minimum sizes (*i.e.*, $|S \cup R| + |S| * |R|$), the minimum size (*i.e.*, $|S \cup R|$), and the average between them (*i.e.*, $|S \cup R| + 0.5 * |S| * |R|$). During generation, we collect the total time, the average size, and the failure number.

The experimental results are shown in Figure 2, where the horizontal axes of all the figures represent the sizes of the grammar's alphabets, the vertical axes 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 size ratios with respect to the required size, and the failure numbers respectively, and "*l-n*" means that the required size is l and the size function $estSize$ is $|S \cup R| + n * |S| * |R|$.

The results show that our algorithm can generate DREGs efficiently: the total times cost by most DREG generations are in several seconds, except the case of 50-0 for the grammars with the alphabet size larger than 15 (due to the underestimating of $estSize$). In particular, when either $|S \cup R| + |S| * |R|$ or $|S \cup R| + 0.5 * |S| * |R|$ is used as the size function, the cost time is in 1s, even in 0.1s when the required size is large (*e.g.* 50).

Moreover, from Figures 2(a)-2(c), we can see that (i) the larger the size function, the fewer (better) the cost time; and (ii) the cost time does not increase rapidly (almost the same) as the alphabet size increases. The reason is that a larger estimated size can trigger the size control mechanism earlier, and thus leave more "space" for the smaller selection.

While considering the actual sizes of the generated DREGs, as shown in Figures 2(d)-2(f), most generations can generate the DREGs such that their average size accounts for more than 60% of the required one. In particular, taking the minimum size as the size function performs best: the average size of the generated DREGs is quite close to the required one. Different from the cost time, the smaller the size function, the larger (better) the average size. The reason is that a smaller estimated size makes the algorithm to generate the DREGs with a closer size to the required one.

Due to the same reason as the cost time, the larger the size function, the less the failure number. Indeed, the failure number contributes to the cost time. Figure 2(i) shows that the failure number is no more than 100, when taking either $|S \cup R| + |S| * |R|$ or $|S \cup R| + 0.5 * |S| * |R|$ as the size function to generate DREGs with size no larger than 50.

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

Experiments on large grammars. We would also like to see how our algorithm performs when generating *long* DREGs from *large* grammars. So we try to generate 100 DREGs with size no larger than 500 from grammars whose alphabet size ranges from 21 to 27. Table 6 shows the experimental results, where $|\Sigma|$ denotes the size of the alphabet Σ, $TTime$ denotes the total time in seconds for the random generation, $Failure$ denotes the failure number during the generation, and $ASize$ denotes the average size of the generated DREGs.

The results show that our algorithm is still effective to generate *long* DREGs from *large* grammars: generating a DREG with size no larger than 500 from a grammar with alphabet size 27 is about $5.833s$ on average. Moreover, as the alphabet size increases, the cost time rises. Meanwhile, thanks to the size function we take, the failure number is very low (almost 0), and the average sizes still account for more than 60% of the required ones.

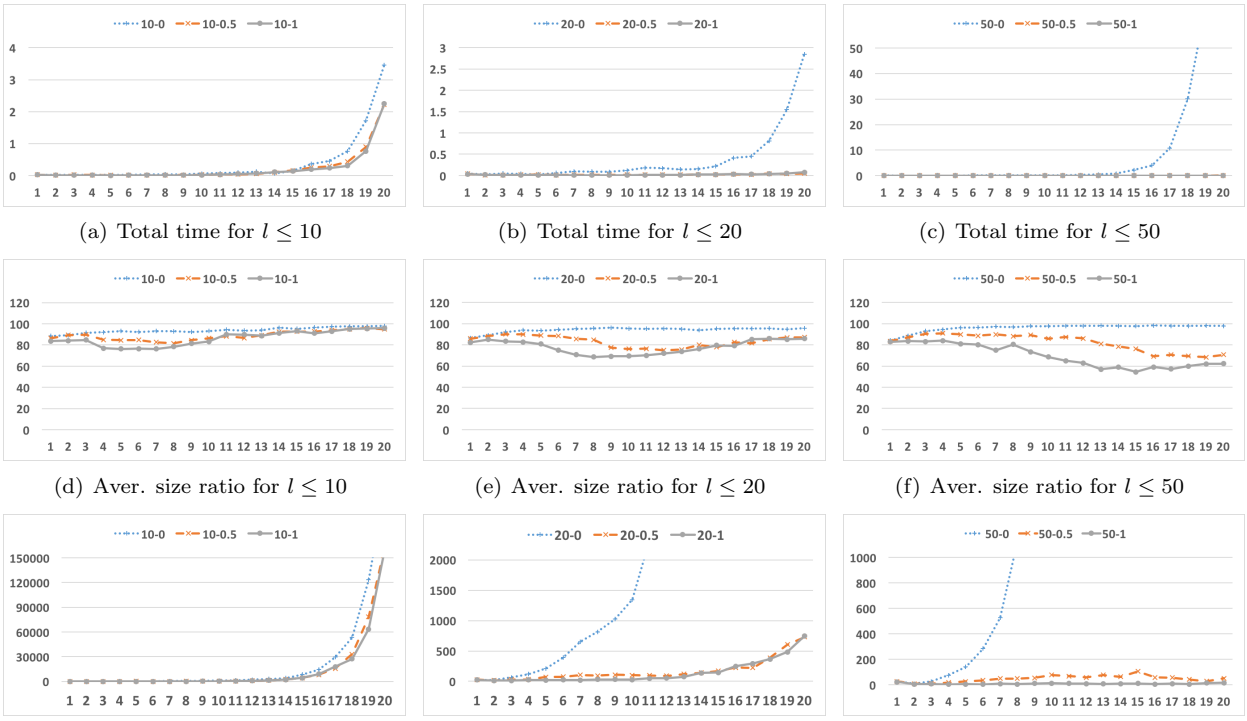
6.3. Comparison with the RE Algorithm

In this section, we present the experiments to compare our random generation algorithm with the RE algorithm (*i.e.*, the algorithm generating regular expressions first and then selecting the deterministic ones). For that, we first conduct experiments to generate DREGs from *the same grammar* with *different sizes*, and then conduct experiments to generate DREGs from *different grammars* with *the same size*, using both our algorithm and the RE algorithm.

The RE algorithm. By RE algorithm, we refer to the one generating regular expressions (RE for short) first with a random sentence generator for RE and then selecting those deterministic ones [9]. To avoid the time caused by the pre-proceeding, we take the simplest random generation algorithm, namely Hanford's algorithm [27], as the RE generator. We also employ the size control mechanism [42] to ensure the termination. In detail, the RE generator takes as input the following grammar for REs:

$$R \rightarrow a_i \mid R\ R \mid R\ o\ R \mid R^+ \mid R\ ? \mid \varepsilon,$$

[8] From 28 on, the generation fails due to the *Java* heap space.



(a) Total time for $l \leq 10$ (b) Total time for $l \leq 20$ (c) Total time for $l \leq 50$

(d) Aver. size ratio for $l \leq 10$ (e) Aver. size ratio for $l \leq 20$ (f) Aver. size ratio 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 size estimations

TABLE 6. Time, average size and failure for generating DREGs of $Len \leq 500$

$\|\Sigma\|$	$TTime$	$Failure$	$ASize$	$\|\Sigma\|$	$TTime$	$Failure$	$ASize$
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				

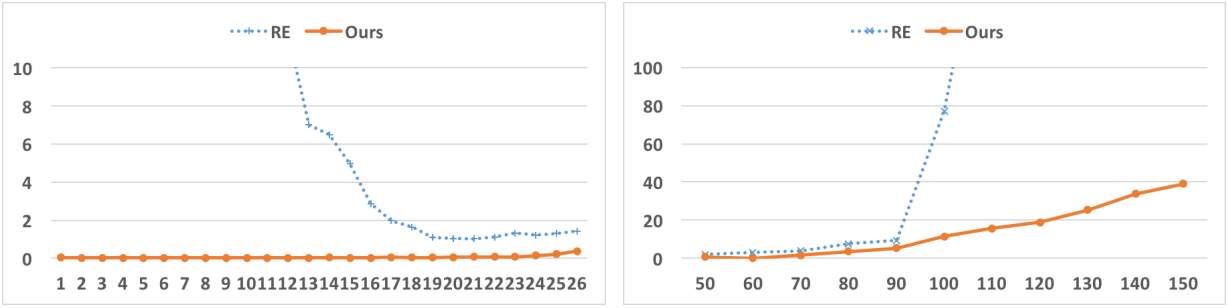
where a_i represents any symbol in alphabet and o denotes the union. The RE generator starts with the nonterminal R and selects a rule for each nonterminal randomly during generation. Meanwhile, the RE generator counts the current size of the generated sentential form. If the current size exceeds the given one, then the RE generator starts to select the terminal rules $R \to a_i$ randomly.

Experiments on different grammars with the same size. In these experiments, we first use our algorithm to respectively generate 100 DREGs with size no larger than 50 from the grammars, whose alphabet size ranges from 1 to 26. Next, using the RE algorithm, we redo each experiment by setting the required size as the corresponding average size of the DREGs generated by our algorithm. The results are shown in Figure 3(a), where the horizontal axis represents the sizes of the alphabets of the grammars, and the vertical axis represents the total times in seconds used by our algorithm and the RE algorithm.

The results show that our algorithm is much more efficient than the RE algorithm, especially for the grammars with small alphabets. There are two factors making the RE algorithm inefficient[9]: (1) one is the very low ratio of DREGs in regular expressions (*e.g*, to generate 100 DREGs with size 50 from grammar with alphabet size 26, we need to generate about 8618 REs with a ratio 1.2%); (2) and the other is the determinism checking, which has to be done for every RE. As our algorithm generates DREGs directly, these two factors are absent, yielding an efficient generator. Moreover, if the required size, rather than the average size from our algorithm, were used for the RE algorithm, then it would cost much more time (see the following experiments). Even worse, the RE algorithm could not terminate in a day for the grammars with small alphabets.

Experiments on the same grammar with different sizes. In these experiments, we use our algorithm on the grammar with alphabet size 26 to generate 100 DREGs with size no larger than a given

[9]If some other algorithms, such as Hickey and Cohen's algorithm [29], were used, the pre-proceeding could be another factor.



(a) Total time for different sizes

(b) Total time for different sizes

FIGURE 3. Comparison between ours and the RE algorithm

size, ranging from 50 to 150. Then we do the same experiments as above. Figure 3(b) shows the results, where the horizontal axis represents the required size on DREGs and the vertical axis is the same as Figure 3(a).

The results show that as the given size increases, the times cost by the RE algorithm increase rapidly. As a result, it cannot generate a DREG with size larger than 100 in an hour. In particular, when the given size is much larger than the alphabet size, it cannot generate a DREG in one day. This is mainly due to the fact that as the given size increases, the probability that the generated regular expression is deterministic decreases very quickly. On the contrary, thanks to the size estimating function, our algorithm still performs well: (i) it generates the required DREGs with size no larger than 50 in 1 second; and (ii) the cost time seems linear in the required size.

The conclusion. The results above 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-proceeding) and the very low ratio of DREGs in RE, we can conclude that our algorithm cannot be replaced by any RE generation algorithm.

6.4. Application of DREG Generator

In this section, we present an application of our DREG generator in the inclusion checker for DREGs in [45].

The inclusion checker presented in [45] implemented two inclusion algorithms, both of which take two DREGs r_1 and 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 sizes are required. However, due to the inefficient generation (*i.e.*, the RE generator), the inclusion checker are evaluated only on DREGs with size no larger than 30. To evaluate the inclusion checker further, here we use our DREG generator to generate longer DREGs from the grammar with alphabet size 26 and use these DREGs to test the inclusion checker. In detail, similar to [45], we first generate 100 pairs of DREGs with size no larger than 50, 100, 150 and 200, and then apply the inclusion checker on these 100 pairs of generated DREGs. Meanwhile, we account the average size of the generated DREGs, the generated time, and the checking time. Table 7 gives the experimental results, where $RSize$ and $ASize$ denote the required size and the average size of the generated DREGs respectively, $SMax$ and $SMin$ denote the maximum and the minimum of the DREG sizes respectively, $GTime$ and $CTime$ denote the times in seconds cost by the generating and by the checking respectively. The results show that the inclusion checker still works effectively on DREGs with size no larger than 200.

TABLE 7. Results for the inclusion checker

$RSIze$	$ASize$	$SMax$	$SMin$	$GTime$	$CTime$
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

This experiment demonstrates that our DREG 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.

7. CONCLUSION

In this paper, we presented syntactic grammars for DREGs. Each production of the grammars is of the form $X \to a_i|\varepsilon$, $X \to Y\ uo$, or $X \to Y\ bo\ Z$, where $a_i \in \Sigma$, uo denotes the unary operators and bo denotes 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 and the syntax of DREGs is context-free.

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. During the generation, it selects the production

of a nonterminal uniformly. Further, it imposes a conservative length condition and a length control mechanism to ensure the efficiency of the generator. Experiments showed that the generator is efficient and useful in practice.

There is several work to do. (1) Further studies of grammar constructions as started in this paper. It is possible to find other ways to construct the grammars, for example using the continuing property [10], in the hope to find some smaller grammars. (2) Extending regular expressions with other operators. In this paper we are considering standard regular expressions. It will be useful to include other commonly used operators, such as counting and interleaving, into the grammars. (3) Optimization rules for the grammars, 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 to help the user writing 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-processing to consider 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 third author was visiting Frank Neven. Neven soon decided to move to other research topics, and he kindly allowed us to freely use and pursue the work initiated during the visit. So we thank Frank Neven for this. We also thank Wim Martens for fruitful discussion about an early version of this paper, and anonymous reviewers for suggestions to improve the presentation of the paper. This work is supported by the National Natural Science Foundation of China under Grants No. 61472405, 61502308 and 61872339, and is partially funded by Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Project 2016050 supported by SZU R/D Fund and Natural Science Foundation of SZU (Grant No. 827-000200).

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. (2013) The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.*, **38**, 24:1–24:39.

[4] Huang, X., Bao, Z., Davidson, S. B., Milo, T., and Yuan, X. (2015) Answering regular path queries on workflow provenance. *ICDE 2015, Seoul, South Korea, April 13-17, 2015*, Los Alamitos, CA, USA, 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'11, Beijing, China, April 18-20, 2011.*, Berlin, Heidelberg, pp. 301–312. Springer.

[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'09, Providence, Rhode Island, USA, June 29 - July 2, 2009*, 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., Bremer, J., and Chen, H. (2015) Deciding determinism of regular languages. *Theory Comput. Syst.*, **57**, 97–139.

[16] Latte, M. and Niewerth, M. (2015) Definability by weakly deterministic regular expressions with counters is decidable. *MFCS'15, Milan, Italy, August 24-28, 2015*, Berlin, Heidelberg, pp. 369–381. Springer.

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

[18] Peng, F., Chen, H., and Mou, X. (2015) Deterministic regular expressions with interleaving. *ICTAC'15, Cali, Colombia, October 29-31, 2015*, Berlin, Heidelberg, pp. 203–220. Springer.

[19] Ahonen, H. (1996) Disambiguation of SGML content models. *PODP'96, Palo Alto, California, USA, September 23, 1996*, Berlin, Heidelberg, pp. 27–37. Springer.

[20] Bex, G. J., Neven, F., Schwentick, T., and Tuyls, K. (2006) Inference of concise DTDs from XML data. *VLDB'06, Seoul, Korea, September 12-15, 2006*, New York, NY, USA, pp. 115–126. VLDB Endowment ACM.

[21] Bex, G. J., Neven, F., and Vansummeren, S. (2007) Inferring XML Schema definitions from XML data. *VLDB'07, University of Vienna, Austria, September 23-27, 2007*, New York, NY, USA, pp. 998–1009. VLDB Endowment ACM.

[22] 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.

[23] Freydenberger, D. D. and Kötzing, T. (2015) Fast learning of restricted regular expressions and DTDs. *Theory of Computing Systems*, **57**, 1114–1158.

[24] 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.

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

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

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

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

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

[31] 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.

[32] McKenzie, B. (1997) Generating strings at random from a context free grammar. Technical Report TR-COSC 10/97. Department of Computer Science, University of Canterbury, Christchurch, New Zealand.

[33] 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*, Basel, Switzerland, pp. 113–125. Birkhäuser.

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

[35] 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, Montral, Canada, Sep 2010.*

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

[37] Héam, P.-C. and Masson, C. (2011) A random testing approach using pushdown automata. *TAP 2011, Zurich, Switzerland, June 30 - July 1, 2011*, Berlin, Heidelberg, pp. 119–133. Springer.

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

[39] Xu, Z., Zheng, L., and Chen, H. (2010) A toolkit for generating sentences from context-free grammars. *SEFM 2010, Pisa, Italy, 13-18 September 2010*, Los Alamitos, CA, USA, pp. 118–122. IEEE Computer Society.

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

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

[42] Zheng, L. and Wu, D. (2009) A sentence generation algorithm for testing grammars. *COMPSAC 2009, Seattle, Washington, USA, July 20-24, 2009*, Los Alamitos, CA, USA, pp. 130–135. IEEE Computer Society.

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

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

[45] Chen, H. and Chen, L. (2008) Inclusion test algorithms for one-unambiguous regular expressions. *ICTAC'08, Istanbul, Turkey, September 1-3, 2008*, Berlin, Heidelberg, pp. 96–110. Springer Berlin Heidelberg.