Generating, sampling and counting subclasses of regular tree languages *

Timos Antonopoulos
Hasselt University and
Transnational University of Limburg
timos.antonopoulos@uhasselt.be

Floris Geerts
University of Edinburgh
fgeerts@inf.ed.ac.uk

Wim Martens
Technische Universität Dortmund
wim.martens@udo.edu

Frank Neven
Hasselt University and
Transnational University of Limburg
frank.neven@uhasselt.be

Abstract

To experimentally validate learning and approximation algorithms for XML Schema Definitions (XSDs), we need algorithms to generate uniformly at random a corpus of XSDs as well as a similarity measure to compare how close the generated XSD resembles the target schema. In this paper, we provide the formal foundation for such a testbed. We adopt similarity measures based on counting the number of common and different trees in the two languages, and we develop the necessary machinery for computing them. We use the formalism of extended DTDs (EDTDs) to represent the unranked regular tree languages. In particular, we obtain an efficient algorithm to count the number of trees up to a certain size in an unambiguous EDTD. The latter class of unambiguous EDTDs encompasses the more familiar classes of single-type, restrained competition and bottom-up deterministic EDTDs. The single-type EDTDs correspond precisely to the core of XML Schema, while the others are strictly more expressive. We also show how constraints on the shape of allowed trees can be incorporated. As we make use of a translation into a well-known formalism for combinatorial specifications, we get for free a sampling procedure to draw members of any unambiguous EDTD. When dropping the restriction to unambiguous EDTDs, i.e. taking the full class of EDTDs into account, we show that the counting problem becomes #P-complete and provide an approximation algorithm. Finally, we discuss uniform generation of single-type EDTDs, i.e., the formal abstraction of XSDs. To this end, we provide an algorithm to generate k-occurrence automata (k-OAs) uniformly at random and show how this leads to uniform generation of single-type EDTDs.

*We acknowledge the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICDT 2011, March 21–23, 2011, Uppsala, Sweden.
Copyright 2011 ACM 978-1-4503-0529-7/11/0003 ...$10.00

Categories and Subject Descriptors

F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages; G.2.1 [**Discrete Mathematics**]: Combinatorics; H.2.1 [**Database Management**]: Logical Design

General Terms

Algorithms, Design, Theory

1. Introduction

XML Schema is the accepted industry standard for the specification of schemas for collections of XML documents. At the same time, it is widely recognized that XML Schema is not a simple language. As it is very unlikely that the World Wide Web Consortium (W3C) will adopt a new schema standard any time soon, several initiatives have been taken to simplify XML Schema. For instance, algorithms have been developed to automatically infer XML Schema Definitions (XSDs) from XML data [8, 9, 10]. We later refer to this setting as the *learning scenario*. Another type of simplification is to let users design a schema in a different, but more user-friendly formalism and then offer the means to automatically convert this schema into an XSD. In general, the latter schema can not be equivalent but, hopefully, constitutes a best approximation in some well-defined way. The latter approach was taken in [18]. We later refer to this setting as the *approximation scenario*. In addition, algorithms to approximate non-deterministc content models by deterministic ones, hereby relieving the user from the Unique Particle Attribution constraint, are studied in [7].

Because it is not always possible to formally prove optimality of the above mentioned types of algorithms, their effectiveness is usually validated by an experimental study using real-world data, for instance using XSDs and corresponding XML corpora found on the web. Unfortunately, as real

world data is often only sparsely available, ad-hoc methods are used to generate schemas and corresponding XML corpora. At the same time, a similarity measure is needed that quantifies how closely two unranked regular tree languages resemble each other, and which can be efficiently computed.

The aim of this paper is to provide the machinery to efficiently compute the similarity between two tree languages and to provide algorithms to generate a corpus of XSDs uniformly at random. As usual, we use the abstraction of XSDs as single-type unranked regular tree languages [28, 30]. In particular, we consider the following three problems:

(i) **Counting:** Given a tree language $\mathcal{L}$ and $n \in \mathbb{N}$, compute the number of trees in $\mathcal{L}$ of size n;

(ii) **Sampling:** Given a tree language $\mathcal{L}$, generate uniformly at random a tree $t \in \mathcal{L}$;

(iii) **Generation:** Given a class of tree languages $\mathcal{C}$, generate uniformly at random a member $\mathcal{L} \in \mathcal{C}$.

We next provide further motivation and describe our contributions for each of these three problems.

Counting and Sampling. We start by discussing an approach towards a similarity measure for tree languages. To this end, let $\mathcal{S}$ and $\mathcal{T}$ be two tree languages. In the schema learning case described above, $\mathcal{T}$ can be the target language and $\mathcal{S}$ can be the schema inferred by the learning algorithm under consideration. Or, in the second scenario of schema approximation, $\mathcal{T}$ can be the schema designed by the user and $\mathcal{S}$ is an approximation of $\mathcal{T}$ in a certain (simple) subclass of tree languages. This raises the natural question of how closely $\mathcal{S}$ resembles $\mathcal{T}$. In this paper, we approach this problem by quantifying the number of common and different trees in $\mathcal{S}$ and $\mathcal{T}$. For instance, one possibility is to define the *similarity* of $\mathcal{S}$ and $\mathcal{T}$ as

$$\mathrm{sim}_{\leq n}(\mathcal{S}, \mathcal{T}) := \frac{\sum_{k=0}^{n} |(\mathcal{S} \cap \mathcal{T})^{=k}|}{\sum_{k=0}^{n} |(\mathcal{S} \cup \mathcal{T})^{=k}|},$$

where the set of trees of size k in a language $\mathcal{L}$ is denoted by $\mathcal{L}^{=k}$, and the cardinality of $\mathcal{L}^{=k}$ is denoted by $|\mathcal{L}^{=k}|$. This similarity measure coincides with a measure commonly used when comparing regular *string* languages [7, 8, 9]. Furthermore, this measure has a natural probabilistic interpretation: the similarity between $\mathcal{S}$ and $\mathcal{T}$ is defined as (an approximation) of the expected probability that a tree, chosen uniformly at random from $\mathcal{S} \cup \mathcal{T}$, belongs to $\mathcal{S} \cap \mathcal{T}$. The approximation is realized by restricting attention to trees up to a certain size n. The algorithmic challenge is to efficiently compute $|\mathcal{L}^{=k}|$ for a tree language $\mathcal{L}$.

For string languages, when $\mathcal{L}$ is represented by a deterministic finite automaton, the counting problem reduces to counting the number of accepting paths in a graph; an easy exercise in dynamic programming. However, when $\mathcal{L}$ is represented by an NFA the problem becomes #P-complete [23]. We establish a similar dichotomy for *tree languages*.

Three classes of unranked regular tree languages are of immediate interest to us: single-type, restrained competition, and bottom-up deterministic EDTDs. Whereas single-type EDTDs correspond to the core of XML Schema [28, 30], restrained competition EDTDs correspond to EDTDs that can be correctly typed in a one-pass pre-order manner [28]. Both of these classes are deterministic in a top-down sense and are strict subclasses of the unranked regular tree languages. Moreover, the single-type EDTDs are known to be a strict subclass of the restrained-competition EDTDs [28]. The class of bottom-up deterministic EDTDs are deterministic in a bottom-up sense and correspond to the full class of unranked regular tree languages. We observe that while every restrained-competition EDTD is equivalent to a bottom-up deterministic EDTD, there is in general no efficient translation. Indeed, in some cases an exponential size increase can not be avoided.

In fact, we consider the class of *unambiguous* EDTDs in which any tree can have at most one valid typing. We observe that every single-type, restrained-competition and bottom-up deterministic EDTD is in effect an unambiguous EDTD. As a consequence, it suffices to develop counting and sampling algorithms for unambiguous EDTDs only. Rather than providing an ad-hoc dynamic programming solution to count the number of trees of a certain size in an unambiguous EDTD, we exhibit a mapping from the class of unambiguous EDTDs into a (recursive) *combinatorial specification*. The latter is a formalism defined by Flajolet, Zimmermann and Van Cutsem [17] and provides an elegant way to derive counting and sampling algorithms. We show that in the case of unambiguous EDTDs, these algorithms are also efficient. In addition, we show how to incorporate *shape constraints* into combinatorial specifications. These are numerical constraints on the depth and width of trees in relation to the total size of the tree. For instance, to avoid string-like trees, we can restrict the depth of a tree to be at most logarithmic in the total number of nodes. In this way, the computation of the similarity of two tree languages can be restricted to trees of a certain shape (which is not necessarily regular).

Finally, when going beyond unambiguous EDTDs, the counting problem becomes intractable. That is, for general EDTDs, we show that computing the number of trees of a certain size is #P-complete. However, we do provide a pseudo-polynomial approximation algorithm based on a similar result for context-free grammars [20].

Generation. To assess the average behaviour of an algorithm, one can test it on a substantial input set drawn uniformly at random. This approach makes sense when no or few real-world data is available and opens up the possibility to quantify the quality of the obtained results in terms of confidence intervals.

In this paper, we consider the problem of generating XSDs uniformly at random. That is, for each n, every non-isomorphic XSD of size n must be generated with the same probability. This definition is the same as for the random generation of deterministic finite automata [2, 5]. Furthermore, since XSDs can be modelled as top-down DFAs that map states to content models [28, 26], we can extend methods for DFA generation to XSDs.

Unfortunately, current DFA generation methods do not constrain the occurrence of alphabet symbols, a constraint important for XSDs. Indeed, it has been noted in [8] that content models in XSDs contain large alphabets but every alphabet symbol occurs only a small number of times. We have referred to such expressions with alphabet symbol occurrence up to k as k-OREs (k-occurrence regular expressions) and to their automata counterparts as k-OAs (k-occurence automata). In this paper, we provide an algo-

rithm to generate uniformly at random deterministic k-OAs and show how this leads to uniform XSD generation.

Outline. In Section 2, we introduce the necessary definitions concerning automata, regular expressions and abstractions of XML schema languages. We study the counting and sampling problem for tree languages in Section 3 and 4, respectively. The uniform generation problem for XSDs is discussed in Section 5. Finally, Section 6 contains related work and the paper is concluded in Section 7.

2. Preliminaries

We define regular expressions, automata and XML Schema languages. First, we fix some basic notation.

2.1 String languages

Strings. For any two integers $n, m \in \mathbb{N}$ where $n \leq m$, we denote by $[n, m]$ the set of all the integers j such that $n \leq j \leq m$. A *symbol* is an element of the alphabet Σ and a *string* w is a finite sequence of symbols $\sigma_1 \cdots \sigma_n$ for some $n \in \mathbb{N}$. We assume that the alphabet Σ is finite. We define the length of a string $w = \sigma_1 \cdots \sigma_n$, denoted by $|w|$, as n and we also refer to $|w|$ as the size of w. The empty string is denoted by ε. If w_1 and w_2 are two strings, we denote their concatenation by $w_1 \cdot w_2$ or simply by $w_1 w_2$. The set of all strings is denoted by Σ^* and a *string language* is a subset of Σ^*. If L_1 and L_2 are two string languages, their concatenation is defined as the set $\{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$, and is denoted by $L_1 \cdot L_2$ or simply by $L_1 L_2$. For a string language L and for any $k \in \mathbb{N}$, we denote by $L^{=k}$ the set of strings in L that have length or size k, namely $L^{=k} = L \cap \Sigma^k$.

Automata. A *non-deterministic finite automaton* (NFA) A is a tuple $(\Sigma, Q, I, F, \delta)$, such that Q is a finite set of states, $I \subseteq Q$ is the set of initial states, F is the set of final states, and δ is the transition function of the automaton, defined as $\delta : Q \times \Sigma \to 2^Q$, mapping each pair of a state and symbol to a set of states. A run ρ of A on some string $w = a_1 \cdots a_n$ is a sequence of states $q_0, \ldots, q_n$, such that $q_0 \in I$ and for each $i \in [1, n]$, $q_i \in \delta(q_{i-1}, a_i)$. Furthermore, when q_n is a member of F, we say that a run is accepting. The string language accepted by A is denoted by $\mathcal{L}(A)$ and is defined as the set of strings w for which there exists an accepting run of A on w. A non-deterministic finite automaton A is said to be *deterministic* (or A is a DFA) if the transition function maps each state/symbol-pair to a *singleton* set.

Regular expressions. The set of *regular expressions* (REs) over Σ is defined recursively as follows. The empty string ε and every symbol in Σ is a regular expression and if r_1 and r_2 are regular expressions, then so are $r_1 \cdot r_2$, $r_1 + r_2$, r^+ and r^*. The string language defined by a regular expression r, is denoted by $\mathcal{L}(r)$ and is defined as follows. If $r = \varepsilon$ then $\mathcal{L}(r) = \{\varepsilon\}$ and if $r = \sigma$ for some $\sigma \in \Sigma$, then $\mathcal{L}(r) = \{\sigma\}$. If $r = r_1 \cdot r_2$ then $\mathcal{L}(r) = \mathcal{L}(r_1)\mathcal{L}(r_2)$, if $r = r_1 + r_2$ then $\mathcal{L}(r) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$, and finally if $r = r_1^+$ then $\mathcal{L}(r) = \{w \mid w = w_1 \cdots w_n$ for some $n \geq 1$ and $\forall i \in [1, n], w_i \in \mathcal{L}(r_1)\}$. For any regular expression r, the regular expression r^* is equivalent to the regular expression $r^+ + \varepsilon$ and $r?$ is used to abbreviate $r + \varepsilon$.

For any regular expression r, we denote by $\bar{r}$ the regular expression obtained from r by replacing, for each i and each $a \in \Sigma$, the i-th occurrence of a by a_i. For example, if $r = ab^*(b + a^+)$, then $\bar{r} = a_1 b_1^*(b_2 + a_2^+)$. The XSD and DTD specifications (to be defined later) restrict regular expressions to be deterministic. A regular expression r is *deterministic* or *1-unambiguous* if there are no strings $w \cdot a_i \cdot v$ and $w \cdot a_j \cdot v'$ in $\mathcal{L}(\bar{r})$ such that $i \neq j$ [14]. We recall that a deterministic regular expression can be translated into an equivalent DFA in quadratic time [12].

2.2 XML schema languages

Trees. A set of strings S is *prefix closed* if for every string $s \in S$ and any prefix s_p of s, s_p is also in S. A *tree* t over an alphabet Σ is a tuple $(\mathsf{Nodes}, \mathsf{lab}, \Sigma)$ where Nodes, the set of nodes of t, is a finite prefix closed set of strings over the natural numbers, such that if $v \cdot i \in \mathsf{Nodes}$ then $v \cdot i' \in \mathsf{Nodes}$ for all $i' < i$, and $\mathsf{lab} : \mathsf{Nodes} \to \Sigma$ is a labelling function assigning symbols of Σ to each node in Nodes. The size of a tree equals its number of nodes. A node $v \in \mathsf{Nodes}$ is a leaf node if there is no $v' \in \mathsf{Nodes}$ such that v is a prefix of v'. The root of t is the empty string in Nodes. The children of a node v in t are all nodes $v' \in \mathsf{Nodes}$ such that $v' = v \cdot i$ for $i \in \mathbb{N}$. The subtree of a tree t at a node v of t is the set of nodes with prefix v. For the tree consisting of a single leaf node v labelled with the symbol σ, we write $\sigma(\varepsilon)$, and for any node v labelled with σ and having subtrees $t_1, t_2, \ldots, t_n$ rooted at its children, we write $\sigma(t_1, t_2, \ldots, t_n)$, denoting the subtree of t rooted at v. The set of all trees over Σ is denoted by Trees_Σ and a tree language $\mathcal{T}$ over Σ is a subset of Trees_Σ. The set of trees over Σ that have exactly k nodes is denoted by $\mathsf{Trees}_\Sigma^{=k}$, for $k \in \mathbb{N}$. For a tree language $\mathcal{T}$, $\mathcal{T}^{=k}$ denotes the set of trees with m nodes, namely $\mathcal{T}^{=k} = \mathcal{T} \cap \mathsf{Trees}_\Sigma^{=k}$.

DTDs and extended DTDs. A DTD over some finite alphabet Σ is a tuple $D = (\Sigma, R, d, S_d)$ where R is a set of deterministic regular expressions over Σ, d is a function that maps symbols in Σ to expressions in R, and $S_d \subseteq \Sigma$ is the set of start symbols. We refer to the regular expressions in R as the *content models* of the DTD. A finite tree t is *valid with respect to* a DTD D or *satisfies* D, if its root is labelled by an element of S_d and, for every node labelled with some $a \in \Sigma$, the sequence $a_1 \cdots a_n$ of labels of its children, is in the language defined by $d(a)$.

A DTD-DFA (Σ, A, d, S_d) over some finite alphabet Σ is a DTD whose content models are represented by the DFAs in the finite set A, instead of regular expressions.

An extended DTD (EDTD) over a finite alphabet Σ is a tuple $(\Sigma, \Delta, R, d, S_d, \mu)$, where Δ is a finite set of types, (Δ, R, d, S_d) is a DTD and μ is a mapping from Δ to Σ. A tree t is *valid with respect to* an EDTD D or *satisfies* D if $t = \mu(t')$ for some tree t' that satisfies the DTD (Δ, R, d, S_d), where μ is extended to trees. We call t' a witness to t.

An EDTD-DFA $(\Sigma, \Delta, A, d, S_d, \mu)$ over a finite alphabet Σ is an EDTD where (Δ, A, d, S_d) is a DTD-DFA.

The tree language consisting of trees that are valid with respect to a DTD or EDTD D is denoted by $\mathcal{L}(D)$. An EDTD D is *reduced* if, for every type τ, there exists a witness tree $t \in \mathcal{L}(D)$ such that the label τ occurs somewhere in t. Any EDTD can be transformed to an equivalent reduced EDTD in polynomial time [1, 27]. In the following, we assume that all EDTDs are reduced.

Let D be an EDTD $(\Sigma, \Delta, d, S_d, \mu)$. Then, for any $\tau \in \Delta$, we denote by D_τ the EDTD $(\Sigma, \Delta, d, \{\tau\}, \mu)$. In particular the set of start symbols S_d of D is changed to $\{\tau\}$. We use

the same notation for EDTD-DFAs.

Subclasses of EDTDs. We recall the following subclasses of EDTDs: *single-type EDTDs*, *restrained competition EDTDs*, and *bottom-up deterministic EDTDs*. Intuitively, these classes have the following significance. Single-type EDTDs are the formal abstraction of XSDs [28] and are therefore central in this paper. The class of restrained competition EDTDs corresponds to the EDTDs that can be correctly typed in a one-pass preorder manner [28]. This means that, when visiting the children of a node from left to right it is clear which type is associated with each node without looking ahead at the nodes to the right. Restrained competition EDTDs form a strict superclass of the single-type EDTDs. Finally, bottom-up deterministic EDTDs are a class of EDTDs that are equally expressive as general EDTDs, i.e., they recognize all regular tree languages. They correspond to bottom-up deterministic tree automata [13].

More formally, let $D = (\Sigma, \Delta, R, d, S_d, \mu)$ be an EDTD.

- D is *single-type* if S_d does not contain two conflicting types and no regular expression in R contains two conflicting types. Here, two types $\tau \neq \tau'$ *conflict* if $\mu(\tau) = \mu(\tau')$.
- D is *restrained competition* if S_d does not contain two conflicting types and all regular expressions in R restrain competition. Here, a regular expression r over Δ *restrains competition* if there are no strings $w\tau v$ and $w\tau' v'$ in $\mathcal{L}(r)$ with $\tau \neq \tau'$ and $\mu(\tau) = \mu(\tau')$.
- D is *bottom-up deterministic*, if for any two distinct types $\tau_1, \tau_2 \in \Delta$, it holds that $\mathcal{L}(d(\tau_1)) \cap \mathcal{L}(d(\tau_2)) = \emptyset$.

These notions are defined analogously for EDTD-DFAs. The class of all single-type (resp. restrained competition,bottom-up deterministic) EDTDs is denoted by By EDTD^{st} (resp. EDTD^{rc}, EDTD^{bud}).

We note that translating between EDTD^{st}s and EDTD^{bud}s gives rise to unavoidable exponential blow-ups. The following proposition holds for all formalisms used for representing content models of EDTDs in this paper.

Proposition 2.1. There is a class $(D_n)_{n\in\mathbb{N}}$ of EDTD^{st}s such that each D_n has size $O(n)$ and the smallest EDTD^{bud} for $\mathcal{L}(D_n)$ has size $2^{\Omega(n)}$. Likewise, there is a class $(D_n)_{n\in\mathbb{N}}$ of EDTD^{bud}s such that each D_n has size $O(n)$ and the smallest EDTD^{st} and EDTD^{rc} for $\mathcal{L}(D_n)$ has size $2^{\Omega(n)}$. □

To conclude this section, we next provide two examples of EDTDs that will also be used in Section 3.

Example 2.2. Consider the EDTD^{st} $D_1 = (\Sigma, \Delta, d, S_d, \mu)$ with $\Sigma = \{a\}$, $\Delta = \{\tau_o, \tau_e\}$, $d(\tau_e) = (\tau_o\tau_o)^*$, $d(\tau_o) = \tau_e(\tau_e\tau_e)^*$, $S_d = \{\tau_e\}$ and $\mu(\tau_o) = \mu(\tau_e) = a$. Then D_1 defines trees of even height where each node at even height has an even number of children and each node at odd height has an odd number of children. Here, the root has height 0. Let D_2 be the EDTD $(\Sigma, \Delta, d', S_d, \mu)$, where d' is such that $d'(\tau_o) = \tau_e(\tau_e\tau_e)^*$ and $d'(\tau_e) = (\tau_o\tau_o\tau_o\tau_o)^*$. Then, D_2 defines trees where a node at odd height has an odd number of children, but nodes at even height have 0 (mod 4) number of children. □

2.3 Unambiguous EDTDs

We next define the class of *unambiguous* EDTDs and show that this class captures the single-type, restrained competition, and bottom-up deterministic EDTDs previously defined.

Definition 2.3. An EDTD or EDTD-DFA is *unambiguous*, denoted by EDTD^{un}, if every tree $t \in \mathcal{L}(D)$ has a unique witness tree t' with $\mu(t') = t$. □

In the remainder of the paper, we regularly use the following observation. The correctness of this observation immediately follows by contraposition.

Observation 2.4. If an EDTD or EDTD-DFA is unambiguous then, for all types $\tau \in \Delta$ and any two distinct trees t_1, t_2 over Δ, if t_1 and t_2 are witnesses to trees in $\mathcal{L}(D_\tau)$, then $\mu(t_1) \neq \mu(\tau_2)$. □

Proposition 2.5. Let $D = (\Sigma, \Delta, d, S_d, \mu)$ be an EDTD. If D is single-type, restrained competition, or bottom-up deterministic then D is unambiguous. □

The following result readily follows from the standard product construction of automata (see, e.g., [18]). We add the observation that, if the input EDTDs are EDTD^{un}s, then the product EDTDs for the union and intersection are also EDTD^{un}s.

Proposition 2.6. Let D_1 and D_2 be two $\text{EDTD-DFA}^{\text{un}}$s. Then we can construct, in quadratic time, an $\text{EDTD-DFA}^{\text{un}}$ for $\mathcal{L}(D_1) \cup \mathcal{L}(D_2)$ and an $\text{EDTD-DFA}^{\text{un}}$ for $\mathcal{L}(D_1) \cap \mathcal{L}(D_2)$. □

Finally, we recall that deciding whether a given EDTD is in of one of the particular classes we use here is in polynomial time.

Proposition 2.7 ([28, 32]). Deciding whether a given EDTD is a EDTD^{st}, EDTD^{rc}, EDTD^{bud}, or EDTD^{un} is in PTIME. □

3. Counting Tree Languages

In this section, we consider the counting problem for tree languages $\mathcal{L}(D)$, where D is an EDTD. More specifically, we show that the counting problem can be efficiently solved when D is unambiguous, even when shape constraints are provided. These results imply that the similarity between unambiguous EDTDs can be efficiently computed. In contrast, the counting problem is shown to be #P-complete for general EDTDs. It does, however, allow for a randomised approximation scheme.

More formally, the general counting problem for languages can be stated as follows:

Definition 3.1. For a class of languages $\mathcal{C}$, given a language $C \in \mathcal{C}$ and $m \in \mathbb{N}$, we define $\#\mathcal{C}$ as the problem of finding the number of members in C of size m. □

For instance, #DFA reduces to counting the number of paths in a graph, while #NFA is known to be #P-complete [23]. Here, we consider #EDTD and $\#\text{EDTD-DFA}^{\text{un}}$ and establish our results by exploiting the close relationship between (unambiguous) EDTDs and derivation trees of (unambiguous) context-free grammars. We first make this relationship precise.

3.1 From EDTDs to CFGs

Recall that a *context-free grammar (CFG)* G is a tuple (N, Σ, R, S) such that N is a finite set of non-terminal symbols, Σ is a set of terminal symbols, R is a subset of $N \times (N \cup \Sigma)^*$ and $S \in N$ is the start symbol. We denote the tuples $(V, w) \in R$ by $V \to w$.

A string $w \in (N \cup \Sigma)^*$ is derived from some non-terminal symbol $V \in N$, denoted by $V \Rightarrow w$, if $V \to w$. The transitive and reflexive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. The language accepted by G is the set of strings $w \in \Sigma^*$ such that $S \Rightarrow^* w$, and is denoted by $\mathcal{L}(G)$. If $T \in N$, then $\mathcal{L}(G_T)$ is the set of strings w such that $T \Rightarrow^* w$. A context-free grammar G is *unambiguous* if, for every string $w \in \mathcal{L}(G)$, w has exactly one derivation tree for G.

It is well-known that regular expressions and (deterministic) finite automata can be translated to equivalent context-free grammars. Here, we use a variation of this result:

Lemma 3.2. Let r be a regular expression and let A be a DFA over alphabet Δ. Then we can construct in linear time a CFG G with start symbol S such that $\tau_1 \cdots \tau_n \in \mathcal{L}(r)$ (resp. $\in \mathcal{L}(A)$) if and only if $S \Rightarrow^* \tau_1 \cdots \tau_n$ in G. Furthermore, if r (resp. A) is deterministic, then G is unambiguous. □

In the remainder of this section, if r is a regular expression, we denote by $\mathrm{CFG}(r, V)$ the set of CFG rules obtained by taking rules of G from Lemma 3.2, replacing the start symbol S by V, and replacing each terminal symbol $\tau \in \Delta$ in the derivation rules by a non-terminal T_τ. That way, we have that $\tau_1 \cdots \tau_n \in \mathcal{L}(r)$ if and only if $V \Rightarrow^* T_{\tau_1} \cdots T_{\tau_n}$ in $\mathrm{CFG}(r, V)$. We define $\mathrm{CFG}(A, V)$ for a DFA A similarly.

Let $D = (\Sigma, \Delta, X, d, S_d, \mu)$ be an EDTD, where X is a set of DFAs or REs. Let $R_{N,\Sigma}$ denote the class of context-free grammar rules over the set of terminal symbols Σ and the set of non-terminal symbols N. Let $\psi_D : \Delta \to \wp(R_{N,\Sigma'})$, for $\Sigma' = \Sigma \cup \{[,]\}$, be a function mapping types to sets of CFG rules, defined as:

$$\psi_D(\tau) = \{T_\tau \to \sigma[R_\tau]\} \cup \mathrm{CFG}(d(\tau), R_\tau)$$

where $\sigma = \mu(\tau)$. In the following, we assume that the non-terminals in the rules $\mathrm{CFG}(d(\tau), R_\tau)$ that are not of the form T_τ, for some $\tau \in \Delta$, are not used elsewhere. However, this can always be achieved by renaming non-terminals accordingly. Let Ψ_D be the set $\{\psi_D(\tau) \mid \tau \in \Delta\} \cup \{S \to T_\tau \mid \tau \in S_d\}$. Notice that, for each type τ there exists exactly one rule in Ψ_D whose left-hand side is T_τ.

The next lemma shows how EDTDs and CFGs are related:

Lemma 3.3. For every EDTD $D = (\Sigma, \Delta, X, d, S_d, \mu)$ the CFG $G_D = (N, \Sigma \cup \{[,]\}, \Psi_D, S)$ is such that for all $n \in \mathbb{N}$, $|\mathcal{L}(D)^{=n}| = |\mathcal{L}(G_D)^{=3n}|$. Furthermore, if D is an EDTD-DFA^{un}, then G is unambiguous. □

Proof Sketch. We show that $|\mathcal{L}(D)^{=n}| = |\mathcal{L}(G_D)^{=3n}|$ by establishing an isomorphism $\mathtt{str}$ between the languages $\mathcal{L}(D)$ and $\mathcal{L}(G_D)$. More precisely, $\mathtt{str} : \mathsf{Trees}_\Sigma \to (\Sigma \cup \{[,]\})^*$ is inductively defined as follows:

$$\begin{aligned} \mathtt{str}(\sigma(\varepsilon)) &= \sigma \cdot [\cdot], \\ \mathtt{str}(\sigma(t_1, \ldots, t_m)) &= \sigma \cdot [\cdot \mathtt{str}(t_1) \cdots \mathtt{str}(t_m) \cdot]. \end{aligned}$$

It then suffices to show that for each n, $\mathtt{str} : \mathsf{Trees}_\Sigma^{=m} \to (\Sigma \cup \{[,]\})^{3m}$ is injective. □

3.2 #EDTD

It readily follows from the hardness of #NFA [23] that #EDTD is hard as well.

Proposition 3.4. #EDTD is #P-complete. □

The relationship between EDTDs and CFGs as specified in Lemma 3.3, however, can be used to provide a randomised approximation scheme for #EDTD.

We recall the notion of randomised approximation schemes for languages from Gore et al. [20]. A *randomised approximation scheme for languages* is a randomised procedure that takes as input a description for a language $L \subseteq \Sigma^*$ and a tolerance $\varepsilon > 0$, and produces as output a number $\hat{L}$ such that $(1+\varepsilon)^{-1}|L| \leq \hat{L} \leq (1+\varepsilon)|L|$ with probability at least $\frac{3}{4}$. For instance, an approximation scheme exists for #CFG:

Theorem 3.5 ([20]). There is a randomized approximation scheme for #CFG, i.e., finding the number of elements of size m of a language defined by a given CFG G, with running time $\varepsilon^{-2}(m|G|)^{O(\log m)}$. □

The approximation scheme for #EDTDs then immediately follows from Lemma 3.3 and Theorem 3.5.

Corollary 3.6. For an EDTD D, there is a randomized approximation scheme for finding the number of elements of size n of the language $\mathcal{L}(D)$, which runs in time $\varepsilon^{-2}(3n|D|)^{O(\log n)}$. □

3.3 # EDTD-DFA^{un}

We again rely on the relationship between EDTD-DFA^{un}s and unambiguous CFGs as specified in Lemma 3.3. That is, we provide an efficient algorithm for $\#\mathcal{L}(G_D)$ (and hence for $\#\mathcal{L}(D)$) by formulating the CFG G_D as a so-called combinatorial specification [17], which we recall below. The advantage of such a specification is two-fold: (1) one can easily bound the complexity of computing the number of objects of a certain size; and (2) one obtains a general sampling procedure for objects in the specification (cf. Section 4)

Combinatorial specifications. A *combinatorial class* is a finite or denumerable set on which a size function is defined, satisfying the following two conditions:

(i) the size of an element is a non-negative integer,

(ii) the number of elements of any given size is finite.

If $\mathcal{A}$ is a class, the size of an element $a \in \mathcal{A}$ is denoted by $|a|$. The set of objects in $\mathcal{A}$ of size n is denoted by $\mathcal{A}_n$. The *counting sequence* of a combinatorial class is the sequence of integers $(A_n)_{n \geq 0}$ where $A_n = |\mathcal{A}_n|$ is the number of objects in class $\mathcal{A}$ that have size n. Two combinatorial classes $\mathcal{A}$ and $\mathcal{B}$ are said to be *combinatorially isomorphic*, written $\mathcal{A} \cong \mathcal{B}$ if and only if their counting sequences are identical. This condition is equivalent to the existence of a bijection from $\mathcal{A}$ to $\mathcal{B}$ that preserves size.

A calculus for combinatorial classes introduced in [17], is presented below. Here, $\mathcal{E}$ and $\mathcal{Z}$ are atoms that denote the classes containing exactly one object of size 0 and size 1 respectively.[1] In the following, we allow different instantiations $\mathcal{Z}_a$, $\mathcal{Z}_b$, ... of the same atom $\mathcal{Z}$. Let $\mathcal{B}$ and $\mathcal{C}$ be combinatorial classes. Then the combinatorial class $\mathcal{A} = \mathcal{B} + \mathcal{C}$

[1] $\mathcal{E}$ is denoted as $\mathbf{1}$ in [17].

is the disjoint union of the classes $\mathcal{B}$ and $\mathcal{C}$. In particular, $\mathcal{Z}+\mathcal{Z}$ contains two objects of size 1. Furthermore, $\mathcal{A} = \mathcal{B}\times\mathcal{C}$ denotes the combinatorial class $\{\alpha = (\beta,\gamma) \mid \beta \in \mathcal{B}, \gamma \in \mathcal{C}\}$ and for each $\alpha = (\beta,\gamma) \in \mathcal{A}$, the size of α is the sum of the sizes of β and γ. For each $\alpha = (\beta_1,\ldots,\beta_n) \in \mathcal{A}$, the size of α is the sum of the sizes of β_i for $i \in [1,n]$. For all n the following hold when $\mathcal{A}$ is a combinatorial class:

$$\begin{array}{ll} \text{if } \mathcal{A} = \mathcal{B}+\mathcal{C} & \text{then } |A_n| = |B_n| + |C_n|, \\ \text{if } \mathcal{A} = \mathcal{B}\times\mathcal{C} & \text{then } |A_n| = \sum_{k=0}^{n} |B_{n-k}| \cdot |C_k| \end{array}$$

In the following, we assume an infinite set of variables C, C_0, C_1, ... Each variable will define a combinatorial class $\mathcal{L}(C)$.

Definition 3.7 ([17]). A *specification* for $(C_1,\ldots,C_n)$ is a collection of n equations, with the i-th equation being of the form

$$C_i := \Psi_i(C_1,\ldots,C_n)$$

where Ψ_i is a term built from $\mathcal{E}, \mathcal{Z}$ and the C_j, using the constructors $+$ and $\times$. For each $j \in [1,n]$, let $C_j^0 = \emptyset$ and for each $i \in \mathbb{N}$, let $C_j^{i+1} = \Psi_j(C_1^i,\ldots,C_n^i)$. Then $\mathcal{L}(C_j)$ is defined to be $\bigcup_{i\geq 0} \mathcal{L}(C_j^i)$. For $k \in \mathbb{N}$, we denote by $\mathcal{L}(C_j)^{=k}$ the objects in $\mathcal{L}(C_j)$ of size k. □

We say that a specification is in *normal form* if each equation is either a single atom, or a single operation $C_i := C_j + C_k$ or $C_i := C_j \times C_k$.

Theorem 3.8 ([17]). Given a specification for $(C_1,\ldots,C_n)$ in normal form and an integer k, the counting sequence up to size k can be computed in $O(n\cdot k^2)$ arithmetic operations. □

From CFGs to combinatorial specifications. We know from Lemma 3.3 that for a given EDTD-DFA^{un} $D = (\Sigma,\Delta,X,d,S_d,\mu)$, the corresponding CFG $G_D = (N, \Sigma\cup\{[,]\}, \Psi_D, S)$ is unambiguous. Furthermore, it is well-known ([17]) that an unambiguous CFG in Chomsky normal form can be translated into a linear-size combinatorial specification $(C_1,\ldots,C_n)$ in normal form by simply replacing concatenation $(\cdot)$ by $\times$, disjunction $(\cup)$ of rules with the same left hand side by $+$, and finally each $\sigma \in \Sigma$ by $\mathcal{Z}_\sigma$. Hence, together with Theorem 3.8 we immediately get:

Theorem 3.9. For an EDTD-DFA^{un} $D = (\Sigma,\Delta,X,d,S_d,\mu)$, the number of trees in $\mathcal{L}(D)$ of size up to k can be computed using $O(|\Delta||Q_{\max}|^2k^2)$ arithmetic operations, where $Q_{\max}$ denotes the largest state space of an automaton in X. □

Proof. It suffices to show that the Chomsky normal form G'_D of the CFG $G_D = (N, \Sigma\cup\{[,]\}, \Psi_D, S)$ has at most $O(|\Delta||Q_{\max}|^2)$ non-terminals. □

We stress that the size of the numbers $|\mathcal{L}(D)^{=k}|$ can grow very fast. To implement the algorithm of Theorem 3.9, a Mathematical Software package is needed. Actually, Maple provides an implementation in the combstruct module of the combinatorial specifications. We implemented our specification for the EDTD-DFA^{un} D_1 given in Example 2.2. As an illustration, we computed the number of trees of size 1001 valid with respect to D_1 and obtained a number with 314 decimals:

5187950237123931732051175236954451756169819365598840423158521214
8190894888949535843265681593434395020810002443582868233520387650
9254373728438806292876525845302947032070990934669778240958562432
2318852268438965431780372366645013594586870608079034900002010371
2015230396579555492265032328755330326988454985168881920847 4

The computation remains under the 60 seconds on a 1.8GHz iMac with 1GB of RAM.

3.4 Shape Constraints

Given an EDTD^{un} $D = (\Sigma,\Delta,A,d,S_d,\mu)$, it is often desirable to count the number of trees in $\mathcal{L}(D)$ that satisfy certain *shape constraints*. Here, by shape constraints we mean certain restrictions on the allowed combinations of the size, depth and/or width of trees in the language. More formally, a shape constraint on the depth (δ) (resp. branching width (w)) of trees consists of a function $\phi_\delta(k)$(resp. $\phi_w(k)$) that assigns to each tree of size k its maximal allowed depth (resp. branching width). For instance, to avoid string-like trees one can take $\phi_\delta(k) = \log k$; to only consider binary trees one simply lets $\phi_w(k) = 2$. As previously described, the counting sequences of trees in $\mathcal{L}(D)$ can be computed using the combinatorial specification corresponding to the CFG G_D. In the presence of shape constraints, we need to augment this specification with parameters corresponding to the depth and width of objects.

We next describe in detail the specification for G_D: Let $D = (\Sigma,\Delta,A,d,S_d,\mu)$ be an EDTD-DFA^{un}. Let $A = \{A_\tau \mid \tau \in \Delta\}$ such that, for each $\tau \in \Delta$, $A_\tau = (\Delta, Q_\tau, \delta_\tau, q_{\tau,0}, F_\tau)$ is the DFA such that $d(\tau) = A_\tau$ and let $Q_\tau = \{q_{\tau,0},\ldots,q_{\tau,m_\tau}\}$. We assume w.l.o.g. that the Q_τ are pairwise disjoint and also disjoint with Δ. Also, let $\mathsf{init} : \{A_\tau \mid \tau \in \Delta\} \to \{q_{\tau,0} \mid \tau \in \Delta\}$ be the function mapping each automaton to its initial state. Finally, we let $Q = \bigcup_{\tau\in\Delta} Q_\tau$.

Given the maximal tree depth d and width w, the specification is defined over the set of variables $\mathbf{Var}(\mathsf{d},\mathsf{w}) = \{R_\tau^{\leq\delta,\leq w}, R_q^{\leq\delta,\leq w}, \mathcal{Z}_{\mu(\tau)} \mid \tau \in \Delta, q \in Q, \delta \in [1,\mathsf{d}], w \in [0,\mathsf{w}]\}$ and is given by the following set of equations:

For $\tau \in \Delta, (\delta,w) \in \mathbb{N}^2, \delta \geq 1$:

$$T_\tau^{(\leq\delta,\leq w)} := \mathcal{Z}_{\mu(\tau)} \times \mathcal{Z}_{[} \times R_{\mathsf{init}(d(\tau))}^{(\leq\delta-1,\leq w)} \times \mathcal{Z}_{]}.$$

For $q \in Q, (\delta,w) \in \mathbb{N}^2, \delta \geq 1$:

$$R_q^{(\leq\delta,\leq w)} := \sum_{\substack{\tau\in\Delta, q'\in Q \\ q'\in\delta_{\tau'}(q,\tau)}} \left(T_\tau^{(\leq\delta,\leq w)} \times R_{q'}^{(\leq\delta,\leq w-1)}\right) \underbrace{+\ \mathcal{E}}_{\text{iff } q\in F_{\tau'}} .$$

We denote by $\mathcal{L}(D)^{(=k,\leq\mathsf{d},\leq\mathsf{w})}$ the set of trees of size k, maximal depth d and maximal width w. A straightforward generalization of Lemma 3.3 then shows that $|\mathcal{L}(D)^{(=k,\leq\mathsf{d},\leq\mathsf{w})}| = |\big(\mathcal{L}\big(\sum_{\tau\in S_d} T_\tau^{(\leq\mathsf{d},\leq\mathsf{w})}\big)\big)^{=3k}|$. Hence, Theorem 3.9 implies:

Corollary 3.10. For an EDTD-DFA^{un} $D = (\Sigma,\Delta,X,d,S_d,\mu)$, size k, depth d and width w, the cardinality of $\mathcal{L}(D)^{(=k,\leq\mathsf{d},\leq\mathsf{w})}$ can be computed using $O(|\Delta||Q_{\max}|^2 \cdot \mathsf{d}\cdot\mathsf{w}\cdot k^2)$ arithmetic operations, where $Q_{\max}$ denotes the largest state space of an automaton in X. □

Proof. This follows immediately from Theorem 3.9. Indeed, it is easily verified that the normalization of the specification $\sum_{\tau\in S_d} T_\tau^{(\leq\mathsf{d},\leq\mathsf{w})}$ contains $O(|\Delta||Q_{\max}|^2\mathsf{d}\cdot\mathsf{w})$ non-terminals. □

We remark that when shape constraints $\phi_\delta(k)$ and $\phi_w(k)$ are provided, $|\mathcal{L}(D)^{(=k,\phi_\delta(k),\phi_w(k))}|$ is easily obtained from Corollary 3.10. Moreover, when only $\phi_\delta(k)$ or $\phi_w(k)$ is provided one simply removes the w or δ parameter, respectively, from the above specification and the complexity is adjusted correspondingly. Finally, we observe that when no shape constraint is specified, the specification reduces to the one for G_D.

3.5 Similarity Measure

We return to computing the similarity between two tree languages as defined in the introduction using the machinery obtained above. Specifically, for tree languages S and T, define,

$$\mathrm{sim}_{\leq n}(S,T) := \frac{\sum_{k=0}^{n} |(S \cap T)^{=k}|}{\sum_{k=0}^{n} |(S \cup T)^{=k}|},$$

where $\frac{0}{0}$ is taken to be 1.

Then we can prove the following result:

PROPOSITION 3.11. Assume S and T are specified as unambiguous EDTD-DFAs. Then for any n, $\mathrm{sim}_{\leq n}(S,T)$ can be computed using $O(|\Delta||Q_{\max,S}|^2|Q_{\max,T}|^2 n^2)$ arithmetic operations, where $Q_{\max,S}$ and $Q_{\max,T}$ denote the largest state space of an automaton in S and T, respectively. □

PROOF. Since $\mathrm{sim}_{\leq n}(S,T)$ requires both $|(S \cap T)^{=k}|$ and $|(S \cup T)^{=k}|$, for $k \in [0..n]$, it suffices to bound the operations needed to compute these quantities. By Proposition 2.6, EDTD^{un}s can be computed for $S \cap T$ and $S \cup T$. Hence, all $|(S \cap T)^{=k}|$ for $k \in [0..n]$ can be computed from the specification of $S \cap T$ using $O(|\Delta||Q_{\max,S}|^2|Q_{\max,T}|^2 n^2)$ operations, where $Q_{\max,S}$ and $Q_{\max,T}$ denote the largest state space of an automaton in S and T, respectively. Indeed, this follows from Theorem 3.9 and the fact that the automata in $S \cap T$ consist of product automata of S and T. Due to trees common to S and T, we cannot use $S \cup T$. Instead, we simply use $|S^{=k}| + |T^{=k}| - |(S \cap T)^{=k}|$ for the counting sequence of the union of S and T. From Theorem 3.9 it follows again that these quantities can be computed up to $k = n$ using $O(|\Delta_S||Q_{\max,S}|^2|n^2)$, $O(|\Delta_T||Q_{\max,T}|^2|n^2)$ and $O(|\Delta||Q_{\max,S}|^2|Q_{\max,T}|^2 n^2)$ operations, respectively. As a consequence, $\mathrm{sim}_{\leq n}(S,T)$ requires $O(|\Delta||Q_{\max,S}|^2|Q_{\max,T}|^2 n^2)$ operations. □

To illustrate feasibility, we used our implementation in Maple to compute $\mathrm{sim}_{\leq 100}(D_1, D_2) = 2.405906249 \cdot 10^{-7}$ taking D_1 and D_2 as defined in Example 2.2. The score was computed in as little as a few seconds.

The above definition of the function sim is just one possibility. A related but more general approach would be to consider a probability distribution p on the natural numbers, and define $\mathrm{sim}_p(S,T)$ as

$$\sum_{n \geq 0} p(n) \frac{|(S \cap T)^{=n}|}{|(S \cup T)^{=n}|} = \sum_{t \in S \cap T} p(|t|) \frac{1}{|(S \cup T)^{=|t|}|}, \qquad (1)$$

where $|t|$ denotes the size of t. This means that $\mathrm{sim}_p(S,T)$ is the expected probability that a tree t drawn from $S \cup T$ with probability $p(|t|)$ belongs to $S \cap T$. So, this measure assigns the same probability to trees of equal size.

4. Sampling tree languages

We next turn to the problem of sampling trees of a certain size in a tree language $\mathcal{L}$ uniformly at random. The sampling procedure closely follows the general uniform sampling methodology for combinatorial classes, as outlined in [17, 19, 25]. The general result for sampling objects of size k in a combinatorial class is as follows:

THEOREM 4.1 ([17]). Any combinatorial specification for $(C_1, \ldots, C_n)$ in normal form has a random generation routine that uses precomputed tables of size $O(nk)$ and achieves $O(nk \log k)$ worst case time complexity. The computation of the tables requires $O(nk^2)$ operations. □

In other words, for any tree language $\mathcal{L}$ that is equivalent to a combinatorial specification, one *automatically* obtains a sampling procedure. This is in particular true for EDTD-DFA^{un}s with and without shape constraints, as we have seen in the previous section.

5. Uniform XSD generation

In this section, we provide an algorithm to generate uniformly at random XSDs of a given size. A first step towards XSD generation is sampling of content models.

5.1 Generating Content Models

Almeida et al. provide a uniform sampler for deterministic connected complete DFAs through a string representation of the automata [2]. We next generalize their approach to deterministic k-OAs, by additionally allowing to parameterize on occurrences of alphabet symbols. While the underlying ideas remain the same, the new parameters introduce a higher level of complexity. Furthermore, we employ the formalism of combinatorial specifications to automatically obtain a generation procedure (cf Section 4).

k-Occurrence Automata As mentioned in the introduction, regular expressions in real-world XSDs can have large alphabets, but each of these alphabet symbols typically occurs only a small number of times.[2] A *k-occurrence regular expression* or *k-ORE* is a regular expression where every alphabet symbol occurs at most k times. For instance, $a \cdot (a+b)^*$ is a 2-ORE. However, we will not consider these regular expressions as such.

The reason is that there is little known on uniform regular expression generation. One approach could be to use for instance the context-free grammar of [24] defining all almost reduced regular expressions. However, this would not exclude different regular expressions defining the same language. Rejection sampling can not be used as there is no notion of minimal regular expression. Therefore, instead of using k-OREs, we turn to the corresponding (but slightly larger) class of k-occurrence automata as defined next. Note that these automata are node labelled.

DEFINITION 5.1. A *k-occurrence automaton* A (k-OA) over Σ is a tuple $(V, E, I, F, \mathsf{lab}, e)$ where V is a finite set of states, $E \subseteq (V \times V)$ is the edge relation, $I \subseteq V$ is the initial set of states, F is the set of final states, $\mathsf{lab} : V \to \Sigma$ is the labeling function, and e is a Boolean which is true when A accepts the empty string. We have the additional requirement that every Σ-symbol labels at most k states. □

[2] Actually, most alphabet symbols occur only once.

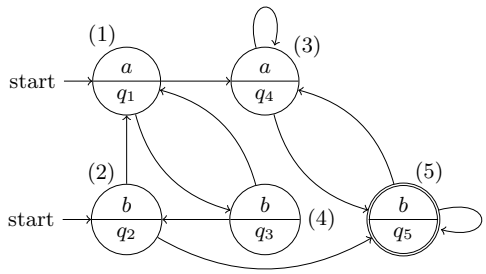


Figure 1: k-OA A

We say that two k-OAs $A_1 = (V_1, E_1, I_1, F_1, \mathsf{lab}_1, e_1)$ and $A_2 = (V_2, E_2, I_2, F_2, \mathsf{lab}_2, e_2)$ are isomorphic, if there exists a bijective function $\beta : V_1 \to V_2$ such that for all $v_1, v_2 \in V_1$, $(v_1, v_2) \in E_1$ if and only if $(\beta(v_1), \beta(v_2)) \in E_2$, $v_1 \in I_1$ if and only if $\beta(v_1) \in I_2$, $v_1 \in F_1$ if and only if $\beta(v_2) \in F_2$, $\mathsf{lab}_1(v_1) = \mathsf{lab}_2(\beta(v_1))$ and $e_1 = e_2$.

A k-OA A is *deterministic* if for every state $s \in V$ and $\sigma \in \Sigma$ there is at most one state $s' \in V$ such that $(s, s') \in E$ and $\mathsf{lab}(s') = \sigma$, and there are no two distinct states $s, s' \in I$ with the same label. A k-OA A is *complete* if for every state $s \in V$ and every $\sigma \in \Sigma$, there exists a state $s' \in V$ such that $\mathsf{lab}(s') = \sigma$ and $(s, s') \in E$, and for every label σ there is a state $s \in I$ with $\mathsf{lab}(s) = \sigma$. Finally, a k-OA A is *connected*, if every state s is reachable from an initial state in I. We call a deterministic, complete and connected k-OA, *admissible*.

String encoding of k-OAs. We next provide a string representation of k-OAs that is inspired by [2]. If A is a deterministic k-OA over Σ with M states, then for every $s \in V$ we denote by $N(s)$ the neighbours of s, which are the states $s' \in V$ such that $(s, s') \in E$. A string encoding, similar to the one introduced in [2], is presented in what follows. This encoding is canonical in the sense that isomorphic automata have the same encoding. For an alphabet Σ of size ℓ, let $o : \Sigma \to [0, \ell - 1]$ be a total order over the symbols of Σ. Given that ordering o, define a canonical total order $c : V \to [1, M]$ as follows. First, for each state $s \in I$ let $c(s) = o(\mathsf{lab}(s)) + 1$. Then, traverse the automaton in a breadth-first way, where at each state s, assign to each of the neighbours of s, that are not yet in the domain of c, and in the order induced by o, the smallest number $n \in [1, M]$ that has not been assigned to a state.

Consider the k-OA A shown in Fig. 1, over the alphabet $\Sigma = \{a, b\}$, where $o(a) = 0$ and $o(b) = 1$, and let $\varepsilon \in \mathcal{L}(A)$. Notice that A is both connected and complete. The canonical total order $c : V \to [1, M]$ is defined as follows. The initial state q_1 with label a is assigned the number 1, and the initial state q_2 is assigned the number 2. Now, traversing the automaton in a breadth-first order, the number 3 is assigned to the state q_4, which is a neighbour of q_1 and has label a. Similarly, the state q_3, which is a neighbour of q_1 with label b is assigned the number 4. Finally, proceeding to the neighbours of q_2, the state q_5 is assigned the number 5 and all states are now ordered. The ordering of the states is annotated between parentheses in Fig. 1.

Given an admissible k-OA A over Σ with M states, the string encoding of A is a string $\mathsf{enc}(A) = S_1 \cdot S_2 \cdot s$ of length $(M + 1) \cdot \ell + M + 1$, where S_1 is the substring encoding the transitions of the automaton and is of length $(M+1) \cdot \ell$, S_2 is the substring encoding the set of final states and is of length M, and finally s is 1 if A accepts the empty string and 0 otherwise. The substring $S_1 = s_0 \cdots s_{(M+1)\cdot\ell-1}$ is such that for each $j \in [0, \ell - 1]$ $s_j = c(i)$ where $i \in I$ and $o(\mathsf{lab}(i)) = j$, and for each $j' \in [1, M]$ and $j \in [0, \ell - 1]$, $s_{j'\cdot\ell+j}$ is equal to $c(v)$ where v is the state of A such that $\mathsf{lab}(v) = o^{-1}(j)$ and $(c^{-1}(j'), v) \in E$. Informally, the latter denotes that $s_{j'\cdot\ell+j}$ is the number corresponding to the state reached from state with number j' reading the alphabet symbol $o^{-1}(j)$. For the substring $S_2 = s_{(M+1)\cdot\ell} \cdots s_{(M+1)\cdot\ell+M-1}$ encoding the set of final states, for each $j \in [0, M - 1]$, $s_{(M+1)\cdot\ell+j}$ is 1 if the state $c^{-1}(j)$ is final and is 0 otherwise.

For example, the string encoding for the k-OA A shown in Fig. 1, is the string

$$\underbrace{12}_{0}\,\underbrace{34}_{1}\,\underbrace{15}_{2}\,\underbrace{35}_{3}\,\underbrace{12}_{4}\,\underbrace{35}_{5}\,\underbrace{00001}_{\text{final}}\,\underbrace{1}_{\varepsilon}.$$

The first two digits of the string encode that state 1 (q_1) is the initial state with label a and that state 2 (q_2) is the initial state with label b. The next two digits encode the outgoing transitions of state 1. The outgoing a-transition goes to state 3 (q_4) and the outgoing b-transition goes to state 4 (q_3). The outgoing transitions for states 2–5 are encoded similarly.

The lemma below assures the correctness of the encoding:

Lemma 5.2. 1. The function c is a bijection for admissible k-OAs.

2. For two admissible k-OAs A_1 and A_2 with n states over an alphabet Σ of size ℓ, with a total order $o : \Sigma \to [0, \ell - 1]$ defined on this alphabet, if $\mathsf{enc}(A_1) = \mathsf{enc}(A_2)$ then A_1 is isomorphic to A_2. □

Characterisation of string encodings. We next provide a characterisation of strings that correspond to encodings of admissible k-OAs. We will leverage upon this characterisation and give a combinatorial specification for these string encodings later on.

For every string $s = s_0 \ldots s_{(n+1)\cdot\ell-1}$, $n \in \mathbb{N}$, we let $(f_i)_{i\in[1,n]}$ be a sequence of numbers such that for each $i \in [1, n]$, f_i denotes the first position in the string s where the number i appears. Then consider the strings $s_0, \ldots, s_{(n+1)\cdot\ell-1}$ that satisfy the following 4 rules:

$$\begin{array}{ll}
(A1) & \forall i \in [1, n-1], f_i < f_{i+1}, \\
(A2) & \forall i \in [1, n], f_i < i \cdot \ell, \\
(A3) & \forall i \in [0, \ell-1], |\{s_p \mid p = i \,(\mathrm{mod}\ \ell)\}| \leq k, \\
(A4) & \forall i, i' \in [0, \ell-1], \text{ where } i \neq i', \\
 & \quad \{s_p \mid p = i \,(\mathrm{mod}\ \ell)\} \cap \{s_p \mid p = i' \,(\mathrm{mod}\ \ell)\} = \emptyset.
\end{array}$$

Intuitively, the above rules express the following: Rule $(A1)$ expresses that for each state i, the first time i appears in the string is before the first time state $i+1$ appears in the string. Rule $(A2)$ expresses that for each state i, the first occurrence of i is in the part of the string encoding the transitions of the first $i - 1$ states, which means that state i is reachable from a state $i' < i$. Rule $(A3)$ expresses that for any symbol σ, there are at most k different states that can be reached by reading σ. Finally, rule $(A4)$ expresses that each state has a unique label.

The precise relationship between the set of strings satisfying $(A1) - (A4)$ and k-OAs is given by the following lemma:

LEMMA 5.3. For each $n \in \mathbb{N}$, enc is a bijection from the set of non-isomorphic admissible k-OAs with n states, over an alphabet Σ of size ℓ to the strings of size $(n+1)\cdot\ell+n+1$ whose prefix $s = s_0, \ldots s_{(n+1)\cdot\ell-1}$ satisfies the rules $(A1) - (A4)$ above, and whose suffix $s' = s_{(n+1)\cdot\ell} \cdots s_{(n+1)\cdot\ell+n}$ uses only 0 and 1. □

In other words, Lemma 5.3 characterizes the strings corresponding to encodings of k-OAs. We next use this characterization to build a combinatorial specification for the set of strings encoding k-OAs.

Combinatorial specification of k-OAs. Let Σ be an alphabet of size ℓ and let A be an admissible k-OA over Σ with n states. Recall that $\mathsf{enc}(A) = S_1 \cdot S_2 \cdot s$ is a string of length $(n+1)\ell + n + 1$, where S_1 is the substring of length $(n+1)\ell$ encoding the transitions of A, S_2 is the substring of length n encoding the set of final states, and finally s is 1 if A accepts the empty string and 0 otherwise. Furthermore, the prefix of S_1 of size ℓ is equal to $1 \cdot 2 \cdots \ell$ and the suffix $S_2 \cdot s$ solely contains 0 or 1. Such fixed strings can easily be combinatorially specified. Indeed, for $i \in [1, n]$, let $\mathcal{Z}_i$ denote the atom corresponding to state i. Then, $\mathcal{Z}_1 \times \cdots \times \mathcal{Z}_\ell$ corresponds to the prefix $1 \cdot 2 \cdots \ell$, whereas for $\mathcal{B} := \mathcal{Z}_0 + \mathcal{Z}_1$, we have that $\mathcal{B}^n \times \mathcal{B}$ corresponds to the set of all possible suffices of size $n+1$ in encodings of k-OAs.

Given these, it remains to specify the remaining symbols in S_1. We need the following notation: For $m \in [\ell+1, n]$, let $\overline{W}^m = [W_0, \ldots, W_{\ell-1}]$ be a partition of $[1, m]$ in which each part is of cardinality at most k. We denote by $\overline{W}^m_{m+1,j}$ the partition of $[1, m+1]$ obtained from $\overline{W}^m$ by adding $\{m+1\}$ to W_j. Finally, if W is a subset of $[1, n]$ then $\mathcal{W}$ denotes the specification $\sum_{i \in W} \mathcal{Z}_i$.

From Lemma 5.3 it follows that the class of strings corresponding to k-OAs of size n can be combinatorially specified as:

$$\mathcal{OA}_n := \mathcal{Z}_1 \times \cdots \times \mathcal{Z}_\ell \times \left(\sum_{p=\ell}^{\ell^2+(\ell-1)} \left(\prod_{i=0}^{p-\ell-1} \mathcal{Z}_{i \,(\mathrm{mod}\ \ell)} \right) \times \mathcal{Z}_{\ell+1} \times S^{(j',j)}_{\ell+1}[\overline{W}^{\ell}_{\ell+1,j\,(\mathrm{mod}\ \ell)}] \right) \times \mathcal{B}^M \times \mathcal{B},$$

where j' and j are such that $p = j'\ell + j$, provided that the class $S^{(j',j)}_{\ell+1}[\overline{W}^{\ell}_{\ell+1,j\,(\mathrm{mod}\ \ell)}]$ consists of all strings of length $n \cdot \ell$ satisfying rules $(A1) - (A4)$, with a fixed prefix $s_0 \ldots s_{j' \cdot \ell + j}$, where $\ell+1$ occurs in the string at position $j' \cdot \ell + j$ for the first time, and for each $i \in [0, \ell-1]$, and $q \in [0, j' \cdot \ell + j]$, it holds that $\{s_q \mid q = i \pmod{\ell}\} = W_i$.

We next claim that the specification given in Figure 2 defines precisely this class of strings. Intuitively, for $\ell < m \leq n$, the equations in the specification reflect that the strings satisfying the rules $(A1) - (A4)$ and in which letter m appears for the first time at position $j' \cdot \ell + j$ is equal to the (disjoint) union of possible strings where $m+1$ appears for the first time in one of the positions between $j' \cdot \ell + j + 1$ and $m \cdot \ell - 1$. In the specification this union is partitioned as follows: The class Q_1 covers the case where between the positions $j' \cdot \ell + j$ and $(j'+1) \cdot \ell + j$ no new state $m+1$ is introduced, whereas the class Q_2 covers the cases where at some position J between the positions $j' \cdot \ell + j$ and $(j'+1) \cdot \ell + j$, the state $m+1$ is introduced. Finally, two cases need special attention: First, in case that $j' = m-1$, the rule $(A2)$ implies that symbol m must appear directly afterwards. In the specification this is reflected by the class defined by Q_3. Second, when $m = n$ and $j' = n-1$, the end of string is nearby and the recursive specification must end. The last equation in the specification deals with this case. Indeed, it simply pads the string with symbols until the desired length of $n\ell$ is reached.

THEOREM 5.4. For $\ell, k, n, j', j, n_0, \ldots, n_{\ell-1} \in \mathbb{N}$ and $m \geq \ell$, the class $\mathcal{S}^{(j',j)}_m[W_0, \ldots, W_{\ell-1}]$ defined by the specification in Figure 2 corresponds to the class of strings of length $(n+1) \cdot \ell$ satisfying rules $(A1) - (A4)$, with a fixed prefix $s_0 \ldots s_{j' \cdot \ell + j}$, where m occurs in the string at position $j' \cdot \ell + j$ for the first time, and for each $i \in [0, \ell-1]$, and $q \in [0, j' \cdot \ell + j]$, it holds that $\{s_q \mid q = i \pmod{\ell}\} = W_i$. □

As a consequence, the specification $\mathcal{OA}_n$ consists of all strings that correspond to the encoding of a k-OA of size n.

LEMMA 5.5. Let $n, k, \ell \in \mathbb{N}$. The number of operations required to compute $|\mathcal{OA}_n|$ is bounded by $O(n^2 \cdot \ell^2 \cdot k^\ell)$. □

PROOF. This is a direct consequence of Theorem 3.8 and the fact that normal form of the specification $\mathcal{OA}_n$ has at most $O(n^2 \cdot \ell^2 \cdot k^\ell)$ non-terminals. □

Uniform generation procedure for k-OAs. Given the specification for k-OAs one can rely on Theorem 4.1 to obtain a uniform random generation procedure. This procedure, referred to as k-$\mathsf{OA\text{-}uniform\text{-}generation}(n, \ell)$, takes as input the size ℓ of the alphabet and the desired number of states n, and returns uniformly at random (with probability $1/|\mathcal{OA}_n|$) a string $\mathsf{enc}(A)$ encoding a k-OA A over Σ and n states.

From Theorem 4.1 and Lemma 5.5 it follows immediately that k-$\mathsf{OA\text{-}uniform\text{-}generation}(n, \ell)$ requires $O(n^2 \cdot \ell^2 \cdot k^\ell n \log(n))$ worst case time complexity.

Finally, we observe that an alternative generation procedure for k-OAs, similar to the one presented in [2], can be devised. The difference is that while generating the string encoding of the admissible k-OAs, one needs to additionally keep track of which alphabet symbols have been assigned to which states, so that no symbol is assigned to more than k states.

5.2 XSD generation

In what follows, we present an algorithm that generates uniformly at random an XSD, whose content models are described by k-OAs, for some $k \in \mathbb{N}$. We make use of the notion of DFA-based XSDs (which is equivalent to XSDs and introduced in [26]) with k-OA content models.

DEFINITION 5.6. A DFA-based k-OA XSD is a tuple $D = (\Sigma, A, \lambda)$, where A is a DFA with set of states Δ over the alphabet Σ, and λ is a function mapping each non-initial state q of A to some k-OA over Σ. A tree t is valid with respect to D if for every node v of t, the sequence $a_1 \cdots a_n$ of labels of its children is in $\mathcal{L}(\lambda(q))$, where q is the state reached by A when started in its initial state, and by reading the string of labels on the path from the root to v. □

For $m \geq \ell$, $j' < m-1$ and $j < \ell$:

$$\mathcal{S}_m^{(j',j)}[\overline{W}^m] := Q_1 + Q_2$$

$$Q_1 := \left(\prod_{i=j+1}^{\ell+j} \mathcal{W}_{i \,(\mathrm{mod}\ \ell)}\right) \times \mathcal{S}_m^{(j'+1,j)}[\overline{W}^m]$$
$$Q_2 := \sum_{i=j+1}^{\ell+j} \left(\prod_{i'=i}^{i-1} \mathcal{W}_{i' \,(\mathrm{mod}\ \ell)}\right) \times \mathcal{Z}_{m+1} \times \mathcal{S}_{m+1}^{(j',j+i)}[\overline{W}^m_{m+1,j+i\,(\mathrm{mod}\ \ell)}]$$

For $m \geq \ell$, $j' = m-1$ and $j < \ell$:

$$\mathcal{S}_m^{(j',j)}[\overline{W}^m] := Q_3$$

$$Q_3 := \sum_{i=j+1}^{2\ell-1} \left(\prod_{i'=i}^{i-1} \mathcal{W}_{j+i' \,(\mathrm{mod}\ \ell)}\right) \times \mathcal{Z}_{m+1} \times \mathcal{S}_{m+1}^{(j',j+i)}[\overline{W}^m_{m+1,j+i\,(\mathrm{mod}\ \ell)}]$$

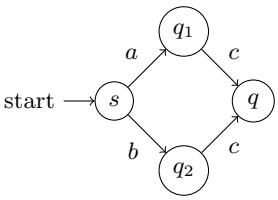
For $m = n$, $j' = n-1$ and $j < \ell$:

$$\mathcal{S}_n^{(n-1,j)}(\overline{W}^n) := \mathcal{W}_{j+1} \times \cdots \times \mathcal{W}_{\ell-1} \times \left(\prod_{i=0}^{\ell-1} \mathcal{W}_i\right)$$

Figure 2: Combinatorial specification of k-OAs.

It therefore suffices to generate DFA-based k-OA XSDs.

Example 5.7. We provide an example of a DFA-based k-OA XSD. Consider the following DFA without final states:



Furthermore, we define $\lambda(q_1) = k\mathrm{OA}(c)$, $\lambda(q_2) = k\mathrm{OA}(cc)$, and $\lambda(q) = k\mathrm{OA}(\varepsilon)$, where $k\mathrm{OA}(r)$ denotes the minimal k-OA for the regular expression r. This DFA-based k-OA XSD accepts two trees, namely $a(c)$ and $b(c,c)$. □

We next discuss the algorithm we need for generating the DFA of a DFA-based k-OA XSD. Example 5.7 illustrates that we need to generate possibly incomplete connected DFAs. Indeed, we cannot simply resort to methods generating minimal DFAs, since the DFA in Example 5.7, having no final states, is obviously not minimal. Furthermore, even when we would define the states that can be reached when reading an entire path from root to leaf in a tree of the language to be final, the resulting DFA is not minimal. Indeed, this approach would define state q to be a final state, but then minimization of the DFA in Example 5.7 would merge states q_1 and q_2, thereby changing the language of the DFA-based XSD. In fact, it can be shown that the DFA-based XSD from Example 5.7 is minimal [29].

This justifies our choice for a method that generates possibly incomplete connected DFAs. More precisely, for $m \in \mathbb{N}$ and any ordered alphabet Σ where $|\Sigma| = \ell$, we consider the algorithm generate-DFA(m, ℓ) from Bassino et al. that generates a possibly incomplete connected DFA with m states (and no final states) over the alphabet Σ [4]. Note that the generated DFAs are not necessarily minimal.

We next show to generate the k-OAs. Whereas we already know to generate k-OAs of a given size, we now need to uniformly range over the sizes of the k-OAs. Let A be a k-OA over an alphabet Σ such that $|\Sigma| = \ell$. Then notice that the number of states for A can only be larger than or equal to ℓ and smaller than or equal to $k \cdot \ell$. For any two values n_1, n_2 such that $n_1, n_2 \in [\ell, k \cdot \ell]$, the number of k-OAs with n_1 states and the number of k-OAs with n_2 states is not necessarily the same, and therefore, to choose uniformly at random the number of states of a k-OA, these numbers have to be taken into account. The algorithm generate-k-OA-num-states relies on the specification $\mathcal{OA}_i$ for k-OAs in order to compute the number of k-OAs with i states over Σ. We next use these numbers to produce a uniform distribution over the number of states for the automaton.

Algorithm 1 generate-k-OA-num-states

Input: ℓ
$\mathsf{max} = \sum_{i=1}^{k \cdot \ell} |\mathcal{OA}_i|$;
return $j \in [1, k \cdot \ell]$ with probability $\frac{|OA_j|}{\mathsf{max}}$

Algorithm 2 generate-k-OA-XSD

Input: m, ℓ
Output: DFA-based XSD (A, λ)
$A =$ generate-DFA(m, ℓ);
for every state $j \in [2, m]$ of A **do**
  $\ell_j = |\{\sigma \mid \delta_A(j, \sigma) \text{ is defined in } A\}|$
  $n_j =$ generate-k-OA-num-states(ℓ_j);
end for
for every state $j \in [2, m]$ of A **do**
  repeat
    $\lambda(j) = k$-OA-uniform-generation(n_j, ℓ_j);
  until is-minimal-complete-k-OA$(\lambda(j))$
end for

Consider then Algorithm 2. The algorithm generate-k-OA-XSD, given m and ℓ, first generates a (possibly incomplete) DFA A with m states over the alphabet $[0, \ell-1]$ uniformly at random. We assume that A's states are numbered from 1 to m and its initial state is 1. Then, for each $j \in [2, m]$, i.e., for each state apart from the initial one, we compute the alphabet size ℓ_j of the k-OA that will be associated to state j. This alphabet size must correspond precisely to the number of outgoing transitions of state j (see Example 5.7)

in order for the DFA-based k-OA XSD to be well-defined. Then we choose the number of states n_j of a k-OA for state j uniformly at random using generate-k-OA-num-states. Finally, we generate a k-OA for state j of A with n_j number of states and alphabet size ℓ_j using k-OA-uniform-generation.

Our aim for Algorithm 2 is to produce, uniformly at random, non-isomorphic *admissible* DFA-based k-OA XSDs, that is, DFA-based XSDs in which the inner DFA is connected and the k-OAs are admissible and minimal.

We note that testing minimality of a deterministic k-OA can be done by a simple adaptation of the standard DFA minimization algorithms. In particular, only states bearing the same alphabet symbol are allowed to be merged.

PROPOSITION 5.8. Testing minimality of a deterministic k-OA can be done in PTIME. □

Therefore, the test is-minimal-complete-k-OA($s_{\text{start,end}}$) runs in polynomial time.

The above discussion leads to the following result.

THEOREM 5.9. Algorithm generate-k-OA-XSD generates, given m and ℓ, uniformly at random an admissible DFA-based k-OA XSD with m types and alphabet size ℓ. □

Finally, we note that the DFA-based k-OA XSDs generated by generate-k-OA-XSD are not necessarily minimal and there can be two such automata A_1 and A_2 generated by the algorithm that generate the same language. We can, therefore, go through a rejection stage that checks if the resulting automaton is minimal, and keeps generating such automata until a minimal one is found. This test whether the generated DFA-based k-OA XSD is minimal can be performed in polynomial time using Proposition 5.8 and the minimization algorithm for XSDs from [29].

As a final remark, we note that generating EDTD^{un}s in a similar manner is computationally harder. Indeed, as an immediate consequence from [11], already testing minimality of a EDTD^{un} is coNP-complete.

6. Related Work

Sampling. Our approach towards counting of tree languages is based on the recursive method, which was initiated by Nijenhuis and Wilf [31], and then formalized by Flajolet, Zimmermann and Van Cutsem [17] in the more general setting of combinatorial specifications. In the current paper we only use a restricted class of specifications (called context-free) in that only atoms, union and product are allowed. General recursive specifications allow many more operators (cf. [17]).

The computational complexity of variants of computing the number of strings of given length in context-free languages is investigated by Bertoni et al. [6]. We choose to employ the method of going through the combinatorial specification as it gives rise to an immediate implementation in Maple and is versatile enough to incorporate shape constraints (which are not context-free definable).

Sampling of trees that adhere to a probabilistic tree model is investigated in [15, 16]. In particular in [16] a sampling procedure for trees that additionally satisfy a bottom-up tree automaton is provided. These methods differ from ours in that trees are sampled in accordance with their probabilities specified by the probabilistic model rather than uniformly.

XSD generation. There has been substantial work on the uniform generation of regular languages represented by DFAs. To the best of our knowledge, no algorithm exists that uniformly generates *minimal* DFAs. The works [2, 5] and [4] do consider the non-isomorphic uniform generation of admissible and connected DFAs, respectively, but need rejection sampling to sample minimal DFAs. This approach, however, has no proven guarantees on running times.

Our string encoding for k-OAs is inspired by [2]. For the generation procedure, however, we rely on a sampling procedure for combinatorial specifications [17]. Although Héam, Nicaud and Sylvain [21] show that non-isomorphic deterministic tree-walking automata can be generated uniformly at random through an encoding into string transducers, it is not clear how to use their results to generate XSDs.

To the best of our knowledge, the present paper presents the first step uniform XSD generation. The papers [9, 8] only dealt with DTDs which reduce to regular expressions. In [10], the experimental validation used one real-world XSD and 8 hand-crafted XSDs. The XSD generation algorithm presented in this paper could be used to generate a benchmark of XSDs. We did not address generation of XML corpora adhering a given schema as is for instance implemented in ToXGene [3].

7. Conclusion

In this paper, we presented a first step towards the foundation for an experimental testbed for XSD generating algorithms. We addressed uniform XSD generation as well as the machinery to compute similarity measures based on counting of trees of a certain size in tree languages. Finally, we provided a sampling procedure for (unambiguous) tree languages using the formalism of combinatorial specifications.

An initial implementation in Maple shows that the approach through combinatorial specifications is promising. Although the approach to assess similarity through counting of the number of different and common trees is intuitive, in depth experimental validation of efficiency and effectiveness remains needed to obtain a concrete robust similarity measure.

Directions for future work include the following:

The complexity of the generation algorithm for k-OAs reveals an exponential behavior in the size of the alphabet. The main reason is that implicitly for each alphabet symbol it needs to be remembered how many times it has already occurred. Fortunately, in real-world content models the far majority of the symbols occur only once. It would be interesting to see how this constraint can be incorporated into the algorithm.

Furthermore, it would be most interesting to extend the XSD generation algorithm to generate k-OREs rather than k-OAs. This would require a useful canonical representation for regular expressions.

Finally, we want to explore the possibility of using specifications (possibly extended with probabilities) to get a non-uniform sampling procedures of trees in a tree language. This would be particularly useful in the probabilistic XML setting, among others.

8. References

[1] J. Albert, D. Giammerresi, and D. Wood. Normal form algorithms for extended context free grammars. *Theoretical Computer Science*, 267(1–2):35–47, 2001.

[2] M. Almeida, N. Moreira, and R. Reis. Enumeration and generation with a string automata representation. *Theoretical Computer Science*, 387(2):93–102, 2007.

[3] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. A. Lyons. ToXgene: a template-based data generator for XML. In *International Symposium on Management of Data (SIGMOD)*, page 616, 2002.

[4] F. Bassino, J. David, and C. Nicaud. Enumeration and random generation of possibly incomplete deterministic automata. *Pure Mathematics and Applications*, 19(2–3):1–16, 2008.

[5] F. Bassino and C. Nicaud. Enumeration and random generation of accessible automata. *Theoretical Computer Science*, 381(1–3):86–104, 2007.

[6] A. Bertoni, M. Goldwurm, and N. Sabadini. The complexity of computing the number of strings of given length in context-free languages. *Theoretical Computer Science*, 86(2):325–342, 1991.

[7] G. J. Bex, W. Gelade, W. Martens, and F. Neven. Simplifying XML Schema: effortless handling of nondeterministic regular expressions. In *International Symposium on Management of Data (SIGMOD)*, pages 731–744, 2009.

[8] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. In *International World Wide Web Conference (WWW)*, pages 825–834, 2008.

[9] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *ACM Transactions on Database Systems*, 2010.

[10] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML Schema Definitions from XML data. In *International Conference on Very Large Data Bases (VLDB)*, pages 998–1009, 2007.

[11] H. Björklund and W. Martens. The tractability frontier for NFA minimization. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 27–38, 2008.

[12] A. Brüggemann-Klein. Regular expressions into finite automata. In *Latin American Symposium on Theoretical Informatics (LATIN)*, pages 87–98, 1992.

[13] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1, april 3, 2001. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[14] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[15] S. Cohen, B. Kimelfeld, and Y. Sagiv. Incorporating constraints in probabilistic XML. *ACM Transactions on Database Systems*, 34(3):1–45, 2009.

[16] S. Cohen, B. Kimelfeld, and Y. Sagiv. Running tree automata on probabilistic XML. In *International Symposium on Principles of Database Systems (PODS)*, pages 227–236, 2009.

[17] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132(2):1–35, 1994.

[18] W. Gelade, T. Idziaszek, W. Martens, and F. Neven. Simplifying XML Schema: Single-type approximations of regular tree languages. In *International Symposium on Principles of Database Systems (PODS)*, 2010.

[19] M. Goldwurm. Random generation of words in an algebraic language in linear binary space. *Information Processing Letters*, 54:229–233, 1995.

[20] V. Gore, M. Jerrum, S. Kannan, Z. Sweedyk, and S. R. Mahaney. A quasi-polynomial-time algorithm for sampling words from a context-free language. *Information and Computation*, 134(1):59–74, 1997.

[21] P.-C. Héam, C. Nicaud, and S. Schmitz. Random generation of deterministic tree (walking) automata. In *International Conference on Implementation and Application of Automata (CIAA)*, pages 115–124, 2009.

[22] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3 edition, 2007.

[23] S. Kannan, Z. Sweedyk, and S. R. Mahaney. Counting and random generation of strings in regular languages. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 551–557, 1995.

[24] J. Lee and J. Shallit. Enumerating regular expressions and their languages. In *International Conference on Implementation and Application of Automata (CIAA)*, pages 2–22, 2004.

[25] H. G. Mairson. Generating words in a context-free language uniformly at random. *Information Processing Letters*, 49(2):95–99, 1994.

[26] W. Martens, F. Neven, and T. Schwentick. Simple off the shelf abstractions of XML Schema. *Sigmod RECORD*, 36(3):15–22, 2007.

[27] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM Journal on Computing*, 39(4):1486–1530, 2009.

[28] W. Martens, F. Neven, T. Schwentick, and G.J. Bex. Expressiveness and complexity of XML Schema. *ACM Transactions on Database Systems*, 31(3):770–813, 2006.

[29] W. Martens and J. Niehren. On the minimization of XML Schemas and tree automata for unranked trees. *Journal of Computer and System Sciences*, 73(4):550–583, 2007.

[30] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, 2005.

[31] A. Nijenhuis and H. Wilf. *Combinatorial algorithms*. Academic Press Inc., 1979.

[32] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal on Computing*, 19(3):424–437, 1990.