Generating XML Structure Using Examples and Constraints

Sara Cohen
The Selim and Rachel Benin School of Engineering and Computer Science
The Hebrew University of Jerusalem
Edmund J. Safra Campus, Jerusalem 91094, Israel
sara@cs.huji.ac.il

ABSTRACT

This paper presents a framework for automatically generating structural XML documents. The user provides a target DTD and an example of an XML document, called a *Generate-XML-By-Example Document,* or a *GxBE document,* for short. GxBE documents use a natural declarative syntax, which includes XPath expressions and the function count. Using GxBE documents, users can express important global and local characteristics for the desired target documents, and can require satisfaction of XPath expressions from a given workload. This paper explores the problem of efficiently generating a document that satisfies a given DTD and GxBE document.

1. INTRODUCTION

Testing is one of most the critical steps of application development. This is certainly correct in the industry, where testing ensures the quality of the end software. It is also correct in academia, where thorough testing can ensure that the proposed solution has a set of properties (e.g., low runtime, improved results versus other systems). For database applications, testing is a challenge both due to the large volume of input data needed, and due to the intricate constraints that this data must satisfy.

The problem of finding or generating input databases for testing is pivotal in the development of database applications. Although such applications are widespread, it is frequently difficult to find or generate appropriate input for testing. This is a significant stumbling block in system development, since considerable resources must be spent to generate test data. In addition, the lack of appropriate test databases is also a bone of contention when studying and comparing academic results. Being able to compare and contrast various academic results is clearly important. This importance has recently been underscored by the experimental repeatability requirements defined by one of the major database conferences (SIGMOD 2008).

To be useful, datasets should have at least the properties:

(1) The data structure must conform to the schema of the target application.
(2) The datasets should be appropriate for the expected workload and have expected characteristics. For example, if the dataset always returns an empty answer on expected queries, it will not be very useful. As another example, we wish to test the application on data with special properties, e.g., a large branching factor.
(3) The data values should should match the expected data distribution.

Currently, there are two distinct sources for XML documents to be used for testing. First, there are several *downloadable XML datasets* that can be found on the web. Such datasets may be real documents of various types, or XML benchmarks, e.g., [12, 13, 15]. Downloadable XML datasets are not always suitable for experimentation and analysis of a particular application or algorithm. Specifically, these datasets may not have any of the three specified properties.

Data generators are a second source for datasets. Several data generators for relational databases have been developed, e.g., [4, 5, 8, 11], while only few XML generators have been developed [1, 2]. Intuitively, an XML generator is a program that generates XML according to given user constraints. XML generators provide the user with flexibility in defining the XML documents of choice. However, their usefulness is determined by the level of expressibility allowed for the users to define the desired target documents. Roughly speaking, [1] produces data that does not satisfy any of the three properties above, since both the structure and the content of the documents are randomly generated. As input, the user may only provide parameters that control the randomness of the result, e.g., the number of levels in the resulting tree and the minimum and maximum number of children for a given level.

ToxGene [2] produces data that satisfies Properties 1 and 3. In [2] constraints are specified locally, within an XML schema. Rich constraints on the values of generated data can be specified, such as data distributions. A simple example of a ToxGene definition appears in Figure 1. A data distribution called c1 is defined, and then is used to define the values that should appear within elements of type my float.

While ToxGene allows rich definition of the data that should be produced, the ability to control the structure of the data defined is very limited (Property 2). All constraints are written locally in the schema. Hence, global properties of the document (e.g., satisfaction of a particular query, a bound on the total document size) cannot be given.

In this paper, we consider an orthogonal problem to that of ToxGene. In particular, we concern ourselves with the

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

```
<tox-distribution name = "c1" type = "exponential"
                  minInclusive = "5" maxInclusive = "100"
                  mean = "35" />
<simpleType name = "my float">
     <restriction base="float">
          <tox-number tox-distribution = "c1">
     </restriction>
</simpleType>
```

Figure 1: A simple ToxGene definition.

```
<state>
   <name>New York</name>
   <city gxbe = "count(↓[store][count(↓[product]) ≥ 400]) ≥ 4"/>
   <city>
      <store gxbe = "count(⇓[discount]) ≤ 0"/>
   </city>
</state>
```

Figure 2: Example GxBE command.

problem of building an XML generator that produces XML with Properties 1 and 2. Thus, the focus is on allowing the user a rich language for defining the *structure* of the target documents. We expect that this work can be combined with that of ToxGene to derive a system that allows rich control of both structure and data. However, such a combination is beyond the scope of this paper.

In our framework, we allow the user to provide a DTD D (to achieve Property 1) and a *Generate-XML-By-Example Document* (*GxBE document* for short) G (to achieve Property 2). A GxBE document is an XML document, with embedded *count constraints*, i.e., constraints expressible in terms of XPath and the aggregation function count.[1] These constraints allow the user to control global properties of the document, as well as to require that various XPath queries be satisfied at least (or at most) a specific number of times. The goal of this work is to efficiently generate a document satisfying both D and G, if such a document exists.

Example 1.1. Consider a company developing an XML-based database application for product inventory. A (simplified) schema for this application might have the form:

$$\begin{aligned} \mathsf{state} &\leftarrow \mathsf{name}, \mathsf{city}^{1..100} \\ \mathsf{city} &\leftarrow \mathsf{name}, \mathsf{store}^{1..5} \\ \mathsf{store} &\leftarrow \mathsf{address}, (\mathsf{email}?, \mathsf{homepage})?, \mathsf{product}^{50..500} \\ \mathsf{product} &\leftarrow \mathsf{name}, \mathsf{price}, \mathsf{discount}? \end{aligned}$$

where $l^{j..k}$ indicates that l must appear at least j, and at most k, times. To shorten the presentation, we omitted definition for labels that simply contain text, e.g., name.

The workload of this application might consist of XPath queries aimed at finding discounted items, checking for available products in a store, etc. Thus, when testing the application, a useful document might containing a store with many discounted items, one with no discounted items, one with many products, one with few, etc. Such requirements can be expressed in a GxBE document. For example, the GxBE command in Figure 2 can be used to generate a document containing at least four stores with many products in one New York city, and one with no discounted products in another New York city. □

The syntax and semantics of GxBE documents, as well as an efficient solution for the document generation problem, are the focus of this paper. Our contributions can be summarized as follows.

- First, we present the language of GxBE documents (Section 2). We find these particularly natural for the document generation problem, since they are "example" XML documents showing what should be generated, and also contain global constraints written in an extension of the standard language XPath.

- Second, we present an efficient algorithm for document generation (Sections 4 and 5). This algorithm is rather intricate, and involves solving additional subproblems, such as determining the number of times that a particular XPath expression can be satisfied in a document. Our algorithm allows many different satisfying outputs to be generated.

- Third, we present several important extensions to the language of GxBE documents (Section 6). For each extension, we show how document generation can still be efficiently achieved, or show that document generation has become intractable.

Due to space limitations, some proofs and subprocedures are omitted. Intuitive explanations of these omissions are provided.[2]

2. FRAMEWORK

In this section we discuss the inputs to our document generator, namely the languages of count-constraints, GxBE commands and DTDs. The syntax that we present allows the more popular XPath operators (i.e., child and descendent axes). In Section 6 we discuss several important extensions of the language of count-constraints, and show how these extensions effect the basic algorithms and the complexity analysis.

2.1 Syntax of Count Constraints

GxBE commands are written using count-constraints. A *count-constraint* c is a numerical constraint over an *XPath expression* p, of the form:

$$\begin{aligned} c &::= \mathsf{count}(p) \geq k, \\ p &::= \downarrow \ \mid \ \Downarrow \ \mid \ p/p \ \mid \ p[l_1, \ldots, l_j] \ \mid \ p[c] \end{aligned}$$

where k is a nonnegative number, and l_i is a label. Intuitively, $\downarrow$ and $\Downarrow$, are, respectively, the child and descendent-or-self axes. We use / to denote the concatenation operator. The qualifier $[l_1, \ldots, l_n]$ constrains the label of the target element. The count-constraint qualifier $[\mathsf{count}(p) \geq k\,]$ (i.e., $[c]$) constrains the number of nodes that satisfy p, relative to the context node. Note that if $k = 1$, then this has the same meaning as a standard nested XPath qualifier.

In Section 6 we will extend the language to allow the operator "$\leq$", as well as horizontal axes (i.e., sibling axes).

[1]Count-constraints first appeared in [7]. The framework there was limited, and document generation was not considered. An algorithm was presented only for constraint satisfiability. This algorithm is simplistic, and regrettably contained a mistake which makes it applicable only for an even more limited constraints.

[2]See also extended version available at `http://www.cs.huji.ac.il/~sara/xmlgen.ps`.

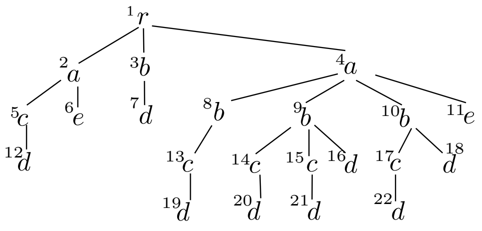

Figure 3: XML Document X_1.

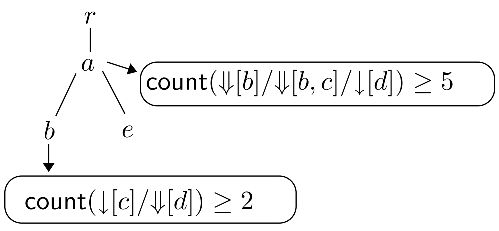

Figure 4: GxBE command G_1.

Intuitively, a count-constraint which does not have any nested count-constraints is simply a requirement that a certain XPath expression be satisfied a number of times. Embedding of count-constraints allows user requirements to be more expressive, in that the number of occurrences of subportions of the XPath expressions may also be bounded from below. This is demonstrated in the following example.

Example 2.1. Consider the following:

$$c_1 = \mathsf{count}(\Downarrow) \geq 1000$$
$$c_2 = \mathsf{count}(\Downarrow/\downarrow[a][\mathsf{count}(\downarrow[b]) \geq 5]) \geq 10$$
$$c_3 = \mathsf{count}(\Downarrow[b]/\Downarrow[b,c]/\downarrow[d]) \geq 5$$

Intuitively, c_1 states that there are at least 1000 nodes in the document. c_2 requires that there be at least 10 a descendents that have at least 5 b children. Finally, c_3 requires that there be at least 5 d descendents that are children of b or c, that is, in turn, a descendent of b. □

2.2 Syntax of GxBE Commands

We use count constraints to specify the properties of interest, that the generated document should satisfy. The user may be interested in several such properties. In addition, he may also have specific requirements for the output document that can be written explicitly (i.e., by explicitly writing fragments of the target document). Both count constraints and explicit constraints may be combined together in an XML document, to form a *GxBE command*. We start by formally defining XML documents, and then we define GxBE commands.

XML documents (or *documents* for short) are rooted, ordered, labeled trees. We will use X to denote documents, and n, n', etc., to denote nodes. We use $\mathrm{lab}(n)$ to denote the label of n and $r(X)$ to denote the root of X. We write (n, n') to denote an edge from n to n' and we write $n \prec n'$ if n appears before n' in an in-order traversal of X. To simplify the presentation, our documents have only elements (and not attributes).

An example of an XML document appears in Figure 3. Only structure, and no data values, is present, since the focus of this paper in on the structure of generated documents. The nodes are numbered for easy reference.

A *GxBE command* G is an XML document, in which each node n is associated with a (possibly empty) set of count constraints $\mathcal{C}(n)$. Technically, these count constraints are written within an XML document as a comma-delimited list of values for the attribute gxbe of a node n. (As before, to simplify the presentation, we assume that only gxbe attributes, and no other attributes, may appear in a GxBE command.)

An example of a GxBE command appeared in Example 1.1. A different example, this time in tree form, appears in Figure 4. The GxBE document G_1 from Figure 4 will be used as a running example in this paper. (Note that G_1 is rather contrived, but this was necessary so that it could be used to demonstrate many different aspects of our algorithm.) Intuitively, G_1 makes the following requirements on the target document: (1) The root of the document should be labeled r. (2) The root should contain at least one child labeled a. (3) There should be at least 5 d descendents that are children of a b or c, that are, in turn, descendents of a b, which are descendents of the a child. (3) In addition, the a child should also have at least the children b and e, in this order. (4) The b child should have at least 2 d descendents that are reachable via c children.

2.3 Semantics of GxBE Commands

In this section, we formally state when a document satisfies a GxBE command. To this end, we first discuss satisfaction of XPath expressions and count constraints.

Let X be an document, let n be a node in X and let p be an XPath expression. We say that X *satisfies* p *at* n if and only if there is a node n' such that $X \models p(n, n')$, defined inductively, in terms of the structure of p, as follows.

- $p =$"$\downarrow$": n' is any child of n
- $p =$"$\Downarrow$": n' is either n, or is a descendent of n
- $p =$"p_1/p_2": there is a node n'' such that a $X \models p_1(n, n'')$ and $X \models p_2(n'', n')$
- $p =$"$p_1[l_1, \ldots, l_j]$": $X \models p_1(n, n')$ and, in addition, $\mathrm{lab}(n') \in \{l_1, \ldots, l_j\}$
- $p =$"$p_1[\mathsf{count}(p_2) \geq k]$": $X \models p_1(n, n')$ and, in addition, $\big|\{n'' \mid X \models p_2(n', n'')\}\big| \geq k$.

Let n be a node in X and p be an XPath expression. Given a count constraint $c = \mathsf{count}(p) \geq k$, we say that X *satisfies* c *at* n, written $X \models c(n)$, if $\big|\{n' \mid X \models p(n, n')\}\big| \geq k$.

Example 2.2. Consider $\mathsf{count}(\downarrow) \geq 3$. This count constraint requires a branching factor of at least 3 and is satisfied by Nodes 1, 4 and 9 in X_1 (Figure 3), since they each have at least 3 children.

As a more complex example, consider Node 4 in X_1. This node satisfies $c_3 = \mathsf{count}(\Downarrow[b]/\Downarrow[b,c]/\downarrow[d]) \geq 5$ (from Example 2.1), since Node 4 has 6 descendents n' (namely, Nodes 19, 20, 21, 16, 22 and 18) such that we have $X_1 \models \Downarrow[b]/\Downarrow[b,c]/\downarrow[d](4, n')$. (We use 4 as a notation for the node numbered 4 in X_1.) □

Finally, consider a GxBE command G. We say that X satisfies G, written $X \models G$, if there is a *generation mapping* μ from the nodes of G to the nodes of X such that:

- *μ maps root to root:* $\mu(r(G)) = r(X)$;
- *μ preserves labels:* $\mathrm{lab}(n) = \mathrm{lab}(\mu(n))$, for all n;
- *μ preserves edges:* if (n, n') is an edge in G, then $(\mu(n), \mu(n'))$ is an edge in X;
- *μ preserves order:* if $n \prec n'$ in G, then $\mu(n) \prec \mu(n')$ in X;
- *μ ensures satisfaction of constraints:* if n is a node in G and $c \in \mathcal{C}(n)$, then $X \models c(\mu(n))$.

EXAMPLE 2.3. Document X_1 satisfies G_1. To see this observe that the following mapping is a generation mapping: $\mu(r) = 1, \mu(a) = 4, \mu(b) = 9, \mu(e) = 11$. (We use r, a, etc., to indicate the node in G_1 labeled r, a, etc.) □

REMARK 1. *The simplest type of GxBE command is an XML document, without count constraints. Such a command is an* example *of desired output, and the result of document generation will be an arbitrary document (conforming to the given DTD) that contains the GxBE command.*

2.4 DTDs

In addition to supplying a GxBE command, the user also supplies a DTD, which states the desired schema of the target document. To be consistent with our presentation, we consider DTDs without attributes. Our DTDs are more expressive than standard DTDs in that we allow labels to be associated with a minimum and maximum number of occurrences.

A *DTD D* is a set of labels $\mathcal{L}$, each of which is associated with an *element description*. Formally, an element description has the form

$$e \ ::= \ \mathcal{S} \ \mid \ l^{j..k} \ \mid \ e, e \ \mid \ e|e \ \mid \ e? \ \mid \ e* \ \mid \ \texttt{empty}$$

We use $\mathcal{S}$ to denote a string value, i.e., PCDATA, and l to denote a label in $\mathcal{L}$. We use $l^{j..k}$ to indicate that l must appear at least j times and at most k times. Note that k can be ∞, in which case the number of times that l appears in unbounded. When $j = k = 1$, we will omit the superscripts. The "," operator as in e_1, e_2 requires that e_1 appears, and then e_2 appears afterwards. The "|" operator is used to denote exclusive disjunction, i.e., $e_1|e_2$ requires that either e_1 or e_2 (but not both) appears. We use $e_1?$ and e_2* to denote that e_1 may appear at most one time, and that e_2 can appear any number of times. Finally, `empty` defines an empty element with no children.

Given a label l, we will use $e(l)$ to denote the element description of l. We will also use the notation $l \leftarrow e$ to indicate that e is the element description of l. Satisfaction of an document X with respect to a DTD D is defined in the obvious way, i.e., *(1)* all the nodes in X must be labeled with labels from $\mathcal{L}$ *and (2)* the children of each node must satisfy the element description of its label.

There are DTDs for which no finite satisfying documents exist. For example, this is the case if the DTD contains the element description $e(r) = r$. Without loss of generality, we will not consider such DTDs in this paper. We note that such unsatisfiable DTDs can be recognized in linear time [3].

We are interested in two special types of DTDs. A DTD is *disjunction-free* if it does not use the operator |. A DTD is *nonrecursive* if all satisfying documents have bounded depth.

EXAMPLE 2.4. The DTD D, for labels $\mathcal{L} = \{r, a, b, c, d, e\}$ appears below.

$$\begin{array}{ll} r \leftarrow (a|b)* & c \leftarrow d \\ a \leftarrow b^{0..4}, c^{0..2}, e & d \leftarrow \mathcal{S} \\ b \leftarrow c^{0..3}, d? & e \leftarrow \mathcal{S} \end{array}$$

Observe that D is nonrecursive, but is not disjunction-free. Let D_1 be the adaption of D in which the definition of r is replaced with the following definition: $r \leftarrow (a?, b?)*$. The DTD D_1 is disjunction-free.

The document X_1 satisfies both D and D_1. □

2.5 Graph Representation of a DTD

For our algorithms later on, it is useful to model a DTD as a graph, consisting of a *Glushkov automaton* for each element definition, and edges connecting these automata. We discuss this modeling.

In the following, we assume that D is a DTD over the labels $\mathcal{L}$ and $l \in \mathcal{L}$. There is a well-known process to transform a regular expression, such as $e(l)$, into a Glushkov automaton [6]. We discuss this process briefly here.

We start by annotating the labels appearing in $e(l)$ so that the same label will not appear twice in the element description. Formally, we use $\hat{e}(l)$ to denote the *annotated version* of $e(l)$ in which the k-th label g is replaced with g_l^k. As an example, consider the DTD D_1 from Example 2.4. Then $e(r)$ is $(a?, b?)*$ and $\hat{e}(r)$ is $(a_r^1?, b_r^2?)*$.

The language of $\hat{e}(l)$ no longer consists of words over the labels in $\mathcal{L}$. We use $\hat{\mathcal{L}}_l$ to denote the labels appearing in the annotated version of $e(l)$, i.e., in $\hat{e}(l)$. For example, $\hat{\mathcal{L}}_r = \{a_r^1, b_r^2\}$.

The Glushkov automaton for l, denoted $\mathcal{A}(l)$, contains the states $\{q_l^0\} \cup \hat{\mathcal{L}}_l$,[3] where q_l^0 is the starting state of $\mathcal{A}(l)$. The edges in $\mathcal{A}(l)$ are defined as follows:

- There is an edge from q_l^0 to each state g_l^k such that there is a word in the language over $\hat{\mathcal{L}}_l$ that satisfies $\hat{e}(l)$ and starts with g_l^k.
- There is an edge from a state g_l^k to a state h_l^j if there is a word in the language over $\hat{\mathcal{L}}_l$ that satisfies $\hat{e}(l)$ and contains the sequence of letters g_l^k, h_l^j.

A state g_l^k is a *final state* if there is a word in the language over $\hat{\mathcal{L}}_l$ that ends with the letter g_l^k. Unlike standard Glushkov automata, we weight the self-loops in our automatons. When such a self-loop results from a definition of the type $l^{j..k}$, we weight the self-loop with k. Other self-loops are weighted with ∞. Intuitively, the weight corresponds to the number of legal consecutive traversals of the loop.

EXAMPLE 2.5. The Glushkov automata for each of the labels in D_1 appear in Figure 5. For the meantime, the dotted arrows should be ignored—they will be discussed later.

Note how optional elements (i.e., elements which may also not appear) create additional edges, e.g, the edge from q_a^0 to c_a^2 in $\mathcal{A}(a)$. Note also how stars give rise to cycles in the automaton, e.g., the cycle between a_r^1 and b_r^2 in $\mathcal{A}(r)$. Finally, the automatons have self-cycles, e.g., that of b_a^1,

[3]We do not make a notational distinction between the labels in $\hat{\mathcal{L}}_l$ and the states in $\mathcal{A}(l)$, since such a distinction would be cumbersome.

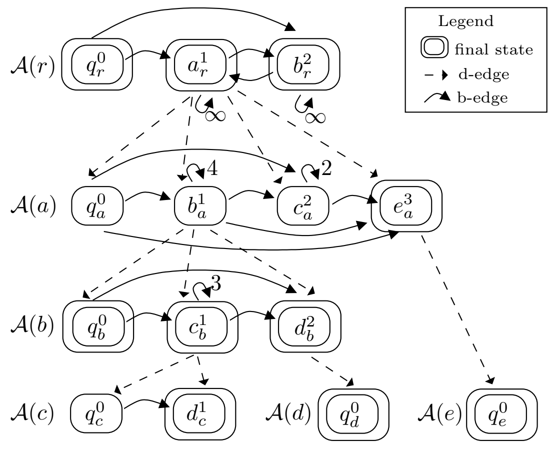

Figure 5: The DTD graph for D_1. Only some of the d-edges have been drawn, to avoid clutter.

which arises from the definition $b^{0..4}$. Observe that this loop has weight 4.

The final states of each automaton are circled. For example, in $\mathcal{A}(r)$ all states are final states, and in $\mathcal{A}(a)$ only e_a^3 is a final state. □

We now define the *DTD graph* $\mathcal{G}(D)$ which contains, as nodes, all states that may appear in the Glushkov automaton of any of the element descriptions, along with some auxiliary nodes. Formally, the nodes of $\mathcal{G}(D)$ are $\hat{\mathcal{L}} = \bigcup_{l \in L} \hat{\mathcal{L}}_l \cup \{l_\perp^1\}$. Note that $l_\perp^1$ is used to simulate an upper level definition in which l is the only element allowed. This corresponds to the outermost nesting level of an XML document.

We now define the edges of $\mathcal{G}(D)$. Consider two states g_l^i and $h_{l'}^j$ in $\hat{\mathcal{L}}$. There are two distinct ways in which these states may be connected:

- $h_{l'}^j$ is a *breadth-child* (or *b-child* for short) of g_l^i if $l' = l$ and there is an edge in $\mathcal{A}(l)$ from g_l^i to $h_{l'}^j$. In other words, b-edges are exactly those edges appearing in the automata, i.e., legal transitions when traversing an XML document breadth-wise.
- $h_{l'}^j$ is a *depth-child* (or *d-child* for short) of g_l^i if $l' = g$, i.e., if $h_{l'}^j$ is any state in $\mathcal{A}(g)$. Intuitively, our d-edges simply correspond to legal transitions when traversing an XML document depth-wise.

In Figure 5 b-edges are drawn as solid arrows and d-edges are dotted arrows. Only some of the d-edges appear, to avoid cluttering the figure.

Example 2.6. Consider $\mathcal{G}(D_1)$, appearing in Figure 5. Observe that a_r^1 and b_r^2 are both b-children of q_r^0. Observe also that all states in $\mathcal{A}(a)$ are d-children of a_r^1. □

We will refer to nodes in $\mathcal{G}(D)$ as states, and use α, β to denote states. If α represents the state g_l^i, then we will use lab(α) to denote g. Thus, for example, if α is c_a^2, then lab(α) $= c$.

A Glushkov automaton is unique in that there is no need to explicitly write a label on any of the edges. An edge from α to β is implicitly labeled with lab(β). We say that a Glushkov automaton is *deterministic* if does not contain two b-edges (α, β) and (α, γ) such that lab(β) = lab(γ). Legal DTDs must be *1-unambiguous*. It is well known that this implies that element descriptions have deterministic Glushkov automatons (that are of size linear in the size of the expressions and can be efficiently generated). This is not an assumption that we make but rather an actual existing requirement of DTDs.

3. PROBLEMS OF INTEREST

We now formally define our main problem of interest, called the *document generation problem*. In the following, we use $X \models (G, D)$ to indicate that X satisfies both G and D.

Problem 1 (Doc. Generation). *Let D be a DTD and let G be a GxBE command. Generate a document X such that $X \models (G, D)$, if such a document exists. Otherwise, output that G and D are mutually unsatisfiable.*

To solve the document generation problem, one needs to create a single document satisfying the constraints. We will discuss some variations of this problem later on, such as finding a bounded-sized document or generating several document satisfying G and D. However, we will soon see that even the problem of creating a single document satisfying G and D is already nontrivial.

Our goal is not only to generate a satisfying document X, but more precisely, we would like to generate X *efficiently*. Unfortunately, we cannot use polynomial time as a yardstick of efficiency, since X may be exponential in the size of G and D, and thus, simply outputting X may take exponential time. To see why, consider the GxBE command G:

<r gxbe="count(⇓) $\geq$ 1000"/>

Any document X satisfying G will be exponentially larger than G, since it will contain at least 1000 nodes (and G uses the number 1000 in binary notation). Even if the numbers in the GxBE command were written in unary notation, X might still be exponential in the size of the input. In particular, this may hold since D can concisely describe a document that is of exponential size, e.g., if D is defined as:

$$\begin{array}{ll} r \leftarrow a_1, a_1 & a_{n-1} \leftarrow a_n, a_n \\ a_1 \leftarrow a_2, a_2 & a_n \leftarrow \mathcal{S} \\ \ldots & \end{array}$$

Any document satisfying this DTD will contain 2^i occurrences of each node a_i.

To overcome this problem, we will adopt input-output complexity. In other words, our aim is to develop a system that returns a document that satisfies both the DTD and the GxBE command in polynomial time in the size of the input and the output.

When measuring the size of the input, we use an adaptation of *data complexity* [14]. In particular, we allow DTDs and GxBE commands to be of arbitrary size. However, we assume that each count constraint in a GxBE command is of bounded size. Note that there may be arbitrarily many count constraints. Note also that we do not assume a bound on the size of the numbers within the count constraints, but rather only on the number of axes. Since we expect count-constraints in a GxBE command to correspond to queries in a workload, assuming boundedness on the constraint size is a natural adaption of the commonly made assumption that queries are small, and thus, bounded.

Unfortunately, even with respect to the complexity class of interest (i.e., input-output complexity, with bounded-size count constraints), the document generation problem may be intractable. This stems from the fact that document generation is strictly harder than the following *satisfiability problem*, which is often difficult in itself. (Note that for satisfiability, we always consider the measure of input complexity, since the output is only a single bit.)

PROBLEM 2 (DOC. SATISFIABILITY). *Let D be a DTD and let G be a GxBE command. Determine whether there exists a document X such that $X \models (G, D)$.*

It is possible for G and D to be contradictory, e.g., if G requires the presence of some label or path not allowed (enough times) by D. Before looking for an efficient solution to the document generation problem, we must rule out certain combinations of DTDs and GxBE commands for which satisfiability is intractable, even when all count constraints are of bounded depth.

Theorem 3.1 shows that satisfiability is intractable, even for very restricted count constraints, if the DTD may contain disjunctions.

THEOREM 3.1. *The satisfiability problem is NP-hard for nonrecursive disjunctive DTDs and GxBE commands, even if the GxBE command does not use the operator $\Downarrow$. If the DTD does not contain any stars ("$*$"), then the satisfiability problem is NP-complete.*

Due to Theorem 3.1, we will only consider DTDs without disjunctions hereafter.

REMARK 2. *Discarding all DTDs which contain disjunctions may seem a bit harsh. Therefore, we note here that there is a common form of disjunction that can easily be eliminated from a DTD, to yield a DTD without disjunction (and thus, which can be used for document generation). This is the case for disjunction of the form $(l_1|\cdots|l_k)*$ which may be equivalently rewritten as $(l_1?, \ldots, l_k?)*$, without changing the set of satisfying documents.*

For example, D from Example 2.4 contains this form of disjunction, and is changed to D_1, as in the above rule. Anecdotally, the DTDs of the DBLP and RSS also contain only disjunction of this form.

4. DOCUMENT GENERATION

In this section we show how to solve the document generation problem, when the DTD is disjunction-free. For this purpose, we present the algorithm GENERATE(G, D, n) (Figure 6). Note that GENERATE uses several sub-procedures that are introduced later on, and thus, will become clearer afterwards. However, we present here an intuitive explanation of GENERATE before drilling down to the details.

The procedure GENERATE considers each node n of G, in an in-order ordering. A document X_1 is created (Lines 1–4) that satisfies all count-constraints associated with n. Then,

```
GENERATE($G, D, n$)
 1  $X_1 \leftarrow$ MAKENODE(lab($n$))
 2  foreach count($p$) $\geq k \in \mathcal{C}(n)$
 3      do $X \leftarrow$ GENSATCON((NF(lab($n$), $\{p\}$), $D$, lab($n$)$^1_\perp$, $k$)
 4         $X_1 \leftarrow$ MERGE($X_1, X, D$)
 5  $X_2 \leftarrow$ MAKENODE(lab($n$))
 6  $(l_1, \ldots, l_m; \phi) \leftarrow$ GENSATWORD($D, n$)
 7  for $i \leftarrow 1$ to $m$
 8      do if $l_i$ is not in the image of $\phi$
 9            then $X \leftarrow$ GENARBITRARYDOC($D, l_i$)
10            else $X \leftarrow$ GENERATE($G, D, n_{\phi^{-1}(i)}$)
11         ADDNEXTCHILD($X_2, X$)
12  return MERGE($X_1, X_2, D$)
```

Figure 6: Algorithm for document generation.

a document X_2 is created (Lines 5–11) that satisfies the constraints implied by the children of n is created. Finally, X_1 and X_2 are merged (Line 12) to derive a document that satisfies both the constraints of n, and the constraints defined by the children of n. To generate a document satisfying G and D, GENERATE should be called with G, D and the root of G.

Now, we review GENERATE with a bit more detail.

Generating a Document Satisfying $c(n)$ (Lines 1–4). Any document satisfying the constraints of n, must be rooted at the label lab(n). We create a node with this label using MAKENODE(lab(n)). Next, we iterate over each count-constraint c associated with n, and create a document satisfying c, by calling the procedure GENSATCON. These documents are merged together using the procedure MERGE. The procedures GENSATCON and MERGE are rather intricate and are discussed in detail later on.

Generating a Document Satisfying the Children of n (Lines 5–11). Once again, we start by creating a node with the label lab(n). Next, we would like to create a series of labels in the language defined by $e(\text{lab}(n))$ that will allow us to satisfy all children of n.

Let $n_1 \prec \cdots \prec n_k$ be the children of n. The procedure GENSATWORD(D, n) returns a series of labels $l_1, \ldots, l_m$, along with a mapping $\phi : \{1, \ldots, k\} \rightarrow \{1, \ldots, m\}$ that satisfies several conditions:

- $l_1, \ldots, l_m$ is a word in the language $e(\text{lab}(n))$;
- ϕ is injective;
- ϕ is consistent with respect to labels, ϕ maps each i to a label equal to lab(n_i);
- ϕ is nondecreasing, i.e., if $i < j$ then $\phi(i) < \phi(j)$.

Intuitively, the first requirement ensures that $l_1, \ldots, l_m$ is in the language defined by the DTD, and the remaining requirements ensure that there will be a generation mapping μ from the nodes of the GxBE command to those of the generated document, as required.

Due to lack of space, we do not provide all details of GENSATWORD. However, it is not difficult to implement this procedure. To see why, let $\mathcal{L}$ be the set of all labels in D. Let $\mathcal{L}_|$ be the disjunction of all labels in $\mathcal{L}$. Consider the regular language defined by the intersection of $e(\text{lab}(n))$

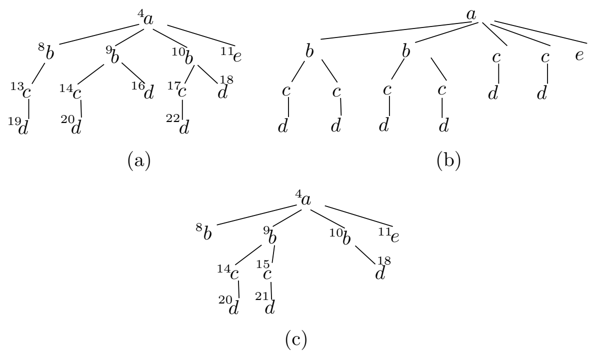

Figure 7: Documents created during Generate.

and

$$(\mathcal{L}_|)*, \mathrm{lab}(n_1), (\mathcal{L}_|)*, \ldots, (\mathcal{L}_|)*, \mathrm{lab}(n_k), (\mathcal{L}_|)* \ .$$

Any word in this intersection is an appropriate return value for GenSatWord.

Finally, we iterate over the labels l_i returned by the procedure GenSatWord. If l_i does not correspond to a child of n, then we generate an arbitrary document rooted at l_i, satisfying D, by calling GenArbitraryDoc.[4] Otherwise, by recursively calling the procedure Generate, we generate a document satisfying the node $n_{\phi^{-1}(i)}$ in G (Line 10). In both cases, these documents are added as children to X_2, using the procedure AddNextChild.

Remark 3. *The procedure* Generate *will work properly and efficiently only if the pair G, D are satisfiable. Thus, this must be checked before calling* Generate.[5]

Remark 4. GenSatWord *and* GenArbitraryDoc *can be implemented in various ways, as long as they return appropriate output. Thus, these algorithms add an element of nondeterminism to* Generate. *We will see later that* GenSatCon *(which generates documents satisfying a constraint) is also nondeterministic, in a fashion. The nondeterminism in our algorithms is useful since it allows us to generate possibly many different documents satisfying the given GxBE command and DTD. For example, if the algorithm* GenArbitraryDoc *creates a random document, then calling* Generate *several times will yield new documents satisfying the inputs.*

Example 4.1. Consider the GxBE command G_1, and the DTD D_1. Suppose that we call Generate with the node in G_1 labeled a. In the first part of the algorithm (Lines 1–4), a document will be create that satisfies the count constraint $\mathsf{count}(\Downarrow[b]/\Downarrow[b,c]/\downarrow[d]) \geq 5$. There are many different documents that can be created. See Figure 7 (a) and (b) for two examples of such documents.

In the second part of the algorithm (Lines 5–11), we must create a word in the language of $e(a)$ that has the labels b and e, in that order. This is because the node labeled a in G_1 has exactly the children b and then e. There are many words that in the language of $e(a)$ that satisfy this requirement, such as *be*, *bbbe*, *bce*, etc. Suppose, for the sake of this example, that GenSatWord returns *bbbe*. The procedure GenSatWord also returns a mapping ϕ that demonstrates how the children of a in G_1 are associated with *bbbe*. There are three different possible mappings in this case, i.e., the node b in G_1 may be associated with any of the three b-s in the word. Suppose that ϕ associates the b from G_1 with the second b in *bbbe*, and ϕ associates the e in G_1 with the e in *bbbe*.

Now, during the loop of Line 7, i iterates from 1 to 4 (the length of *bbbe*). When $i = 1$, an arbitrary document rooted at b is returned, since the first b is not in the image of ϕ. When $i = 2$, a recursive call will be made with Generate to generate a document satisfying all constraints of b in G_1, since the second b was in the image of ϕ. When $i = 3$, another arbitrary document rooted at b will be created. Finally, when $i = 4$, the procedure Generate will be called again for the node e in G_1. Thus, at the end of the second part of the algorithm, the document in Figure 7 (c) may have been generated.

Finally, the documents created in the first two stages of the algorithm will be merged. If Figure 7 (a) and Figure 7 (c) were generated in the first and second stages, respectively, the result will be the subtree rooted at Node 4 of X_1 (Figure 3).

Note that the document X_1 can itself be produced by Generate, when called with the root of G_1. This particular document would be returned if (1) the subtree rooted at Node 4 is produced to satisfy the a node of G_1 (in the first stage of the algorithm), (2) the word *aba*, with mapping ϕ that maps the a of G_1 to the third a in *aba* is produced (in the second stage of the algorithm) and (3) the subtrees rooted at Nodes 2 and 3 are returned by calls to GenArbitraryDoc. □

It can be shown that the algorithm Generate has the following properties.

Theorem 4.2. *Given a disjunction-free DTD D and a GxBE command G,* Generate *returns a document X such that $X \models (D,G)$ (if one such document exists) in polynomial time in the size of the input and output, if each count constraint in G is of bounded size. Moreover, any document X with the above properties can be returned by* Generate, *if the appropriate choices are made.*

The proof of Theorem 4.2 is omitted. However, the intuition behind its correctness will be discussed below, and in the following section.

The two most difficult aspects of Generate that must still be clarified are:

- *Issue 1:* How are documents merged, i.e., how does the Merge procedure (Lines 4 and 12) work? Note merging is nontrivial since we must ensure that the resulting document still satisfies the DTD and constraints.
- *Issue 2:* How can we generate a document that satisfies a given constraint, i.e., how is the procedure GenSatCon of Line 3 actually implemented?

The first issue is addressed briefly below. The second issue is quite intricate, and is the subject of the next section.

[4]This algorithm is very easy to implement. The details of this algorithm are omitted due to lack of space.

[5]We omit the details on how to check satisfiability, since the basic procedure is very similar to that needed for document generation.

Merging Documents. Given documents X_1 and X_2, and a DTD D, the procedure MERGE(X_1, X_2, D) returns a document X such that

- X satisfies D;
- X *contains* X_1 and X_2. In particular, it is possible to remove subtrees from X to derive X_1 (or X_2).

For example, as discussed above, the subtree rooted at Node 4 of X_1, is a merge of Figure 7 (a) and Figure 7 (c).

The count constraints considered thus far are *monotonic*. Intuitively, this means that if c is satisfied by the root of X_1, then it is satisfied by the root of any document containing X_1 (and rooted at the same node as X_1). Thus, after merging documents X_1 and X_2, we derive a document that satisfies all constraints satisfied by X_1 or X_2.

Due to space limitations, we do not give all details of the procedure MERGE. However, there are two questions that immediately arise in regard to MERGE that we answer.

- *Does there always exist a document X containing X_1 and X_2 that conforms to D?* Note first that the document X must be rooted at the same label as X_1 and X_2. Thus, such a document can exist only if X_1 and X_2 have the same label at their roots. This is indeed the only case in which MERGE is called.

  Given that X_1 and X_2 have the same label at their roots, the answer to our question is positive, since D is disjunction-free. Disjunction-free regular expressions have the property that given two words in a language, it is always possible to find a third word in the language that contains both words.

  Note that if D had disjunctions, then there may very well not be a result for merge, e.g., if D had the element definition $r \leftarrow a|b$, then it would not be possible to merge two documents, one containing a and one containing b.

- *How do we find a document X that contains X_1 and X_2?* Having established that such a document exists, we still must be able to find X. In our case, this is not difficult, due to the fact that all element definitions correspond to *deterministic* Glushkov automata (as discussed above). A deterministic Glushkov automaton is unique in the following manner. Let X_1 be a document. Then, there is a unique state $\alpha \in \hat{\mathcal{L}}$ that corresponds to each node $n \in X$. This state can easily be found. The ability to find such states allows MERGE to be defined in a fashion that is very similar to a standard algorithm for merging sorted lists.

5. GENERATING A DOCUMENT SATISFYING A COUNT-CONSTRAINT

We now consider the problem of generating a document that satisfies a given constraint, i.e., the implementation of GENSATCON. To simplify the presentation, we assume in this section that there are no nested count qualifiers and that the DTD is nonrecursive. Both restrictions can be lifted, as discussed at the end of this section. To further simplify the presentation, we will assume without loss of generality that each axis has exactly one set of labels as a qualifier, i.e., that our count-constraints have the form

$$\mathsf{count}(A_1[L_1]/A_2[L_2]/\cdots/A_n[L_n]) \geq k$$

```
GENSATCON($P, D, \alpha, k$)
 1  if $\epsilon \in P$
 2      then $k \leftarrow k - 1$
 3  if $k \leq 0$
 4      then return GENARBITRARYDOC($D$, lab($\alpha$))
 5  $X \leftarrow$ MAKENODE(lab($\alpha$))
 6  Choose a legal path $\beta_1, \ldots, \beta_m$ in $\mathcal{A}$(lab($\alpha$)) and
          numbers $k_1, \ldots, k_m$ such that:
              (1) $0 \leq k_i \leq$ MaxSat($D$, lab($\beta_i$), NFC(lab($\beta_i$), $P$)),
              (2) $\sum_{i=1}^{m} k_i = k$
 7  for $i \leftarrow 1$ to $m$
 8      do $X' \leftarrow$ GENSATCON(NFC(lab($\beta_i$), $P$), $D$, $\beta_i$, $k_i$)
 9          ADDNEXTCHILD($X, X'$)
10  return $X$
```

Figure 8: An algorithm that generates a document satisfying a constraint $\mathsf{count}(p) \geq k$. Should be called with GENSATCON(NF(l, $\{p\}$), D, $l^1_\perp$, k)**.**

where $A_i \in \{\downarrow, \Downarrow\}$ and L_i is a set of labels.[6]

To ease our notation, given an XPath expression $p = A_1[L_1]/A_2[L_2]/\cdots/A_n[L_n]$, we will use Prefix($p$) to denote $A_1[L_1]$ and Suffix(p) to denote $A_2[L_2]/\cdots/A_n[L_n]$. For the special case that $n = 1$, we write Suffix(p) $= \epsilon$, where ϵ is a special symbol denoting the *empty path*.

The procedure GENSATCON appears in Figure 8. The goal of this procedure is to create a document, rooted at a specified label, that satisfies a given XPath expression a specified number of times. For reasons that will become clearer later, GENSATCON is defined in a more general fashion, and does not get a single XPath expression p, but rather a set P of XPath expressions. The goal is to create a document that has at least a given number of nodes, each of which satisfies at least one $p \in P$.

GENSATCON receives four parameters: a set of XPath expressions P, a DTD D, a state α in the DTD graph, and the number of nodes k that should satisfy expressions in P. When we call GENSATCON from GENERATE we desire to create a document rooted at the label lab(n) of a GxBE node n, satisfying a count constraint $\mathsf{count}(p) \geq k$. We provide the initial parameters:

- set of XPath expressions NF(lab(n), $\{p\}$): This contains p, and additional XPath expressions, discussed in detail later;
- the DTD D;
- the state lab$(n)^1_\perp$, which is used (as discussed in Section 2.5) to simulate an upper level root containing a node with label l;
- the number k.

Intuitively, the algorithm runs as follows. First, we check if the empty path ϵ is in P. If so, then the root node created will certainly satisfy P one time, and we decrease k by 1. If $k \leq 0$, then there is no requirement to satisfy P. Hence, an arbitrary document can be generated and returned (Line 4).

Otherwise, if $k > 0$ (Lines 5–10), we must produce a document that satisfies expressions in P at least k times. For this purpose, we start by creating a node with lab(α). This

[6]The set L_i may contain all labels in the DTD, if no stricter constraint is desired.

node will be the root of the document that we create. Next, we choose the children of the root, by choosing a *legal path* in the automaton $\mathcal{A}(\text{lab}(\alpha))$. A legal path is any path that starts at a node that is a neighbor of the starting state, reaches a final state, and does not traverse any self-loop more times consecutively, than the number appearing on the self-loop. Every legal path in $\mathcal{A}(\text{lab}(\alpha))$ corresponds to a word in $e(\text{lab}(\alpha))$ and vice-versa. We will be creating a subtree for each state β_i in the legal path chosen.

Not every legal path will enable us to satisfy P a total of k times. Thus, we must make sure that the legal path chosen will allow this. For this purpose, when we choose our legal path $\beta_1, \ldots, \beta_m$, we also choose numbers $k_1, \ldots, k_m$ such that $\sum_{i=1}^{k} = k$ and it is possible to create a subtree rooted at β_i, while satisfying P at least k_i times. (Formally, this is written as $k_i \leq \text{MaxSat}(D, \text{lab}(\beta_i), \text{NFC}(\text{lab}(\beta_i), P))$ and is discussed in detail below.) Intuitively, the numbers k_i are used to "break up" the value k to determine how the k satisfying nodes will be distributed among the subtrees represented by $\beta_1, \ldots, \beta_m$. Then, a recursive call is made to GenSatCon to create subtrees rooted at the label of β_i, with k_i nodes satisfying P.

We now consider the algorithm in more detail. The issues that must be clarified are as follows:

- Why does GenSatCon receive a set of XPath expressions P, instead of a single XPath expression? Recall that the goal of GenSatCon is to create a document satisfying a single XPath expression p at least k times. Thus, it might not yet be clear why GenSatCon gets a set of XPath expressions, and which sets exactly are considered.

- How do we determine the maximum number of times that an XPath expression can be satisfied by a subtree, i.e., what is $\text{MaxSat}(D, \text{lab}(\beta_i), \text{NFC}(\text{lab}(\beta_i), P))$, and how is it calculated?

We discuss each of these issues, in turn, below.

Sets of XPath Expressions. Recall that our goal is to create a document, rooted at a label l, that satisfies a count-constraint $\mathsf{count}(p) \geq k$. In other words, we desire to create a document X, rooted at a node $r(X)$, with label l, such that X contains (at least) k nodes n for which $p \models (r(X), n)$.

Sometimes, satisfying subexpressions of p at a label l immediately imply that p itself is satisfied. For example, consider the XPath expression $p = \Downarrow[b]/\Downarrow[b, c]/\downarrow[d]$. In order to satisfy p in a document rooted at b, one may have a descendent n_1 labeled b, with a descendent n_2 labeled b or c, with a child n_3 labeled d. Observe that finding descendent nodes n_1 and n_2 is not strictly necessary, since the root itself is labeled b, and therefore the root satisfies the first two steps of p.

To formalize the above reasoning, we define the l-normal form of a set of XPath expressions. Given a set P, the *l-normal form* of P, denoted $\text{NF}(l, P)$, is derived by repeatably applying the following rule while applicable:

If $p \in P$ and $\text{Prefix}(p) = \Downarrow[L]$ where $l \in L$,
Then add $\text{Suffix}(p)$ to P.

We say that P is in l-normal form if $\text{NF}(l, P) = P$.

For example, $\text{NF}(b, \{\Downarrow[b]/\Downarrow[b, c]/\downarrow[d]\})$ is the set containing the three XPath expressions

$$\Downarrow[b]/\Downarrow[b, c]/\downarrow[d] \qquad \Downarrow[b, c]/\downarrow[d] \qquad \downarrow[d]$$

As another example, $\text{NF}(d, \{\Downarrow[d, e]\}) = \{\Downarrow[d, e], \epsilon\}$.

When we first call GenSatCon, to produce a document rooted at a label l satisfying p at least k times, we provide the set $\text{NF}(l, \{p\})$. This is because it is sufficient to satisfy any XPath expression in $\text{NF}(l, \{p\})$ at least k times. Formally, this follows from the following result.

Proposition 5.1. *Let X be a document rooted at a node labeled l and P be a set of XPath expressions. Let n be a node in X. Then,*

$$\exists p \in P \text{ s.t. } p \models (r(X), n) \iff \exists p \in \text{NF}(l, P) \text{ s.t. } p \models (r(X), n)$$

During the algorithm GenSatCon, we recurse on children of the current state. Intuitively, we wish to find k nodes satisfying P, by generating children and then finding descendents of these children that will satisfy P. During our recursion, we must pass to the algorithm the subexpressions that must be satisfied by children nodes, in order for the expressions in P to be satisfied by the root. Thus, given a child label l, we define $\text{NFC}(l, P)$ as the set $\text{NF}(l, S_1 \cup S_2)$ where

$$S_1 = \{p \in P \mid \text{Prefix}(p) = \Downarrow[L]\}$$
$$S_2 = \{\text{Suffix}(p) \mid p \in P \wedge \text{Prefix}(p) = \downarrow[L] \wedge l \in L\}$$

Satisfying any expression in $\text{NF}(l, S_1 \cup S_2)$ at a child l, will imply satisfaction of an expression of P at the parent node. As an optimization step, we remove from $\text{NFC}(l, P)$, all path expressions with prefix $\Downarrow[L]$ for which l has no d-reachable descendent in D that is in L, as well as all path expressions with prefix $\downarrow[L]$ for which l has no d-child in L. These are path expressions which will never be satisfied by any descendent of L.

Example 5.2. Consider the set $P_1 = \{\Downarrow[b]/\Downarrow[b, c]/\downarrow[d]\}$. For the GxBE G_1, we would like to satisfy this constraint 5 times at a node labeled a. For this purpose, we will have to create children for the a node, and satisfy appropriate variations of P_1 at each of these children. Recall that a is defined as $b^{0..4}, c^{0..2}, e$. Observe that $\text{NFC}(b, P_1) = \{\Downarrow[b, c]/\downarrow[d], \downarrow[d]\}$, since we will compute $\text{NF}(b, P_1)$, i.e.,

$$\{\Downarrow[b]/\Downarrow[b, c]/\downarrow[d],\ \Downarrow[b, c]/\downarrow[d],\ \downarrow[d]\},$$

and then discard the first expression since b has no descendents in D_1 that have label b. We also have that $\text{NFC}(c, P_1) = \text{NFC}(e, P_1) = \emptyset$.

Let $P_2 = \text{NFC}(b, P_1)$. Observe that $\text{NFC}(c, P_2) = \{\downarrow[d]\}$ and $\text{NFC}(d, P_2) = \{\epsilon\}$. □

The Value MaxSat. Our algorithm for generation needs the ability to determine the maximum number of times that an XPath expression can be satisfied. Formally, let D be a DTD, l be a label and P be a set of XPath expressions. We define $\text{MaxSat}(D, l, P)$ as follows:

$$\max\{\text{CntSat}_X(r(X), P) \mid X \models D,\ l(r(X)) = l\},$$

where $r(X)$ is the root of X and

$$\text{CntSat}_X(n, P) ::= \left|\{n' \mid \textstyle\bigvee_{p \in P} X \models p(n, n')\}\right|.$$

$$\begin{tikzcd}
\boxed{\mathrm{MaxSat}(D_1, a, \{\Downarrow[b]/\Downarrow[b,c]/\downarrow[d]\}) = 16} \arrow[d] \arrow[dr] \arrow[dd] & \\
\boxed{\mathrm{MaxSat}(D_1, c, \emptyset) = 0} & \boxed{\mathrm{MaxSat}(D_1, e, \emptyset) = 0} \\
\boxed{\mathrm{MaxSat}(D_1, b, \{\Downarrow[b,c]/\downarrow[d], \downarrow[d]\}) = 4} \arrow[d] \arrow[dr] & \\
\boxed{\mathrm{MaxSat}(D_1, c, \{\downarrow[d]\}) = 1} \arrow[r] & \boxed{\mathrm{MaxSat}(D_1, d, \{\epsilon\}) = 1}
\end{tikzcd}$$

Figure 9: Pictorial representation of computing $\mathrm{MaxSat}(D_1, a, \{\Downarrow[b]/\Downarrow[b,c]/\downarrow[d]\})$**.**

Intuitively, $\mathrm{MaxSat}(D, l, P)$ is the maximum number of nodes that can satisfy at least one of the path expressions in P in any document X that satisfies D and is rooted at l. Since there may be infinitely many such documents X, there may be no maximum to the above set, i.e., it may be possible to satisfy P an arbitrary number of times. In such a case, we write $\mathrm{MaxSat}(D, l, P) = \infty$. Note that usually

$$\mathrm{MaxSat}(D, l, P) \neq \sum_{p \in P} \mathrm{MaxSat}(D, l, \{p\}),$$

since the same node may satisfy several XPath expressions, but it is only counted once in the expression of the left-hand side.

Example 5.3. Consider the sets

$$P_1 = \{\Downarrow[b,c]/\downarrow[d]\} \qquad P_2 = \{\Downarrow[b]/\Downarrow[b,c]/\downarrow[d]\}$$

and $P_3 = P_1 \cup P_2$.

Suppose we wish to compute $\mathrm{MaxSat}(D_1, a, P_i)$ for $i \leq 3$, where D_1 is from Example 2.4. We have $\mathrm{MaxSat}(D_1, a, P_1) = 18$. To see this, note that there can be 4 d children of b descendents (since there can be 4 b descendents of a, and each can have one d child). In addition, there can be 14 d children of c descendents, since there can be 2 c children for a and $4 \times 3 = 12$ c descendents of a (and each c can have a single d). The value of $\mathrm{MaxSat}(D_1, a, P_2)$ is lower, and is only 16, since c children of a no longer produce d nodes that satisfy the XPath expression. Finally, $\mathrm{MaxSat}(D_1, a, P_3) = 18$.

As a final example, observe that $\mathrm{MaxSat}(D_1, r, P_3) = \infty$ since there may be arbitrarily many d descendents that satisfy the XPath expressions, due to the "$*$" in $e(r)$. □

Now, computing $\mathrm{MaxSat}(D, l, P)$ proceeds as follows. Let $\mathcal{A}(l)$ be the Glushkov automaton for l. Consider a legal path $\vec{\beta} = \beta_1, \ldots, \beta_k$. Let l_i be $\mathrm{lab}(\beta_i)$. The *weight* of this path, denoted $w(D, P, \vec{\beta})$, is simply the sum

$$\sum_{i=1}^{k} \mathrm{MaxSat}(D, l_i, \mathrm{NFC}(l_i, P)).$$

Let B_l be the set of all legal paths in $\mathcal{A}(l)$.

The following lemma states that $\mathrm{MaxSat}(D, l, P)$ is exactly the maximal weight of any legal path in $\mathcal{A}(l)$, if $\epsilon \notin P$ and is one more than this value if $\epsilon \in P$. (The extra value of 1 is derived since the root itself is a node satisfying P.)

Lemma 5.4. *Let P be a set of XPath expressions in l-normal form. Let D be a DTD. Then,*

$$\mathrm{MaxSat}(D, l, P) = \begin{cases} \max\{w(D, P, \vec{\beta}) \mid \vec{\beta} \in B_l\} & \epsilon \notin P \\ \max\{w(D, P, \vec{\beta}) \mid \vec{\beta} \in B_l\} + 1 & \epsilon \in P \end{cases}$$

Lemma 5.4 provides us with a method to compute the value $\mathrm{MaxSat}(D, l, P)$. In particular, if D is nonrecursive, then $\mathrm{MaxSat}(D, l, P)$ can be computed by reducing to computation at d-child labels. (We discuss the recursive case at the end of the section.) We note that it is not necessary to explicitly consider all legal paths, in order to find the maximum. The largest weight can be found using a simple graph traversal. Roughly speaking, if there is a state β which is in a b-cycle with an edge marked ∞ or with an unmarked edge, for which $\mathrm{MaxSat}(D, b, \mathrm{NFC}(b, P)) \geq 1$, then $\mathrm{MaxSat}(D, l, P) = \infty$. Otherwise, the path with the greatest weight can be found by traversing the graph of $\mathcal{A}(l)$ according to the indices of the states. Exact details are omitted due to lack of space.

Example 5.5. Figure 9 presents a pictorial representation of computing $\mathrm{MaxSat}(D_1, a, \{\Downarrow[b]/\Downarrow[b,c]/\downarrow[d]\})$. The values are computed by a simple analysis of the legal path with greatest weight, along with the values at children nodes, e.g., the value of $\mathrm{MaxSat}(D_1, b, \{\Downarrow[b,c]/\downarrow[d], \downarrow[d]\})$ is 4, since the value at c and d are each 1, and b can have 3 c children and one d child.

There are two important things to note: First, we may compute the value MaxSat more than one time for the same label, each time with a different set P. This occurs for the label c. Second, we may need the same computation several times during the total evaluation. This occurs with $\mathrm{MaxSat}(D_1, d, \{\epsilon\})$ which is needed twice. We only compute each value once, by using memoization to avoid repeated computations. □

Throughout the execution of GenSatCon, every XPath expressions in the argument of MaxSat is always a subexpression of the expression p originally appearing in the GxBE command. In the worst case, we must compute MaxSat for each subset of subexpressions of p, at each label l. By our assumption that XPath expressions are of bounded length, the computation of MaxSat can be performed in polynomial time.

After explaining all components of GenSatCon, we demonstrate its execution.

Example 5.6. In our running example, the procedure GenSatCon will be called for the node a in G_1, with the state $a^1_\perp$, the set of path expressions $\{\Downarrow[b]/\Downarrow[b,c]/\downarrow[d]\}$, along with the value 5. This procedure can return various results, depending on how the legal path and numbers are chosen in Line 6 in GenSatCon.

Consider the document in Figure 7 (a). For this document to be returned, GenSatCon must choose the legal path

$$b^1_a, b^1_a, b^1_a, e^3_a$$

in $\mathcal{A}(a)$, along with the numbers 1, 2, 2, 0. A recursive call will be made with each of the four states in the path.

For the first b^1_a, the recursive call will be made with the number 1 and the set of XPath expressions $\{\Downarrow[b,c]/\downarrow[d], \downarrow[d]\}$. If GenSatCon then chooses the legal path c^1_b with number

1, we will end up returning the subtree of Node 8 (in Figure 7 (a)) for the first b_a^1. Similar call for the remaining states in the legal path will allow us to create exactly Figure 7 (a). □

It remains to relax the two assumptions made at the beginning of the section, i.e., that there are no nested count qualifiers and that the DTD is nonrecursive. We discuss how computation of MaxSat can proceed if there are nested count qualifiers and D is recursive. The extensions to GenSatCon to deal with these cases are similar.

• **Recursive DTDs:** As presented, we compute MaxSat for a particular label l and set P by recursively computing MaxSat at child labels l' of l. However, this may imply an infinite execution if the DTD is recursive (since two labels may be both ancestors and descendents one of another). We observe that such infinite execution may occur only if we are computing MaxSat when P contains only path expressions that begin with $\Downarrow$. (Otherwise, the recursive calls must change P, and shorten the path expressions, and eventually terminate.)
By careful analysis one can derive that there are two possible cases. First, it is possible that for all $p \in P$, there is no way to generate a node satisfying p. Second, it is possible that there is a $p \in P$ for which a descendent can be generated that satisfies p. In the former case, the value 0 can be immediately returned. In the latter case, due to the recursiveness of the DTD, we may satisfy p infinitely many times, and thus ∞ should be returned. By careful preprocessing it is possible to differentiate between these two cases and immediately return 0 or ∞, as required.

• **Nested Count Qualifications:** Consider XPath expressions that contain nested count qualifiers. In this case MaxSat can be computed in a bottom-up fashion. Specifically, given a nested qualifier $[\mathsf{count}(p') \geq k']$ in a path expression p, we first compute $\text{MaxSat}(D, l', \text{NF}(l', \{p'\}))$, for every label l'. Then, we can replace $[\mathsf{count}(p')\theta k']$ with $[l_1, \ldots, l_j]$ where l_i are the labels for which

$$\text{MaxSat}(D, l_i, \{p'\}) \geq k' ,$$

i.e., the labels for which the nested count qualification was satisfied. Hence, by recursively computing MaxSat for nested count qualifiers, we will eventually derive an XPath expression with no nested count qualifiers.

Using the ideas presented in this section, the following result can be shown.

Theorem 5.7. *Given a disjunction-free DTD D and a count constraint c,* GenSatCon *returns a document X satisfying D such that $X \models c(r(X))$ (if one exists) in polynomial time in the size of the input and output, if c is of bounded size. Moreover, any document X with the above properties can be returned by* GenSatCon*, if the appropriate choices are made.*

6. EXTENSIONS

The language of GxBE commands considered thus far allows for XPath expressions with the standard operators $\Downarrow$ and $\downarrow$, along with the operator $\geq$. In this section we consider important extensions to our language and show how these extensions affect the tractability of document generation.

6.1 Constraints with "$\leq$"

It is useful for the number of nodes satisfying an XPath expression to be bounded from above (and not only from below). In other words, we would like to allow count constraints of the form $\mathsf{count}(p) \leq k$. See Example 1.1, which demonstrates the usefulness of such constraints.

Unfortunately, it is not always possible to allow "$\leq$" and remain with a tractable algorithm, as shown in the following theorem.

Theorem 6.1. *The satisfiability problem is NP-hard for disjunction-free nonrecursive DTDs and GxBE commands G if G may contain a node which has:*

- *count constraints with both nested count qualifiers and* $\leq$ or
- *count constraints with both $\geq$ and $\leq$ (even without nested count qualifiers).*

We say that a GxBE command G is *legal* if
1. G does not contain a count constraint with both nested count qualifiers and $\leq$,
2. every node n in G that contains count constraints with $\leq$ is a leaf, and
3. there do not exists nodes n, n' such that n is an ancestor of n' (or $n = n'$) and n has count-constraints with $\geq$, while n' has count-constraints with $\leq$.

Thus, we can have non-empty constraint sets for two nodes on the same path only if the constraints use $\geq$. For example, the GxBE command of Figure 2 is legal.

For legal GxBE commands, we can show the following result, by adapting Generate.

Theorem 6.2. *Given a disjunction-free DTD D and a legal GxBE command G,* Generate *returns a document X such that $X \models (D, G)$ (if one such document exists) in polynomial time in the size of the input and output, if each count constraint in G is of bounded size.*

The basic idea is that one can create a document satisfying a count constraint with $\leq$ by performing the following two steps. First, compute the *minimal version* D_{min} of D by removing all subexpressions of the form $e*$, $e?$ or $l^{0..k}$ and replacing all remaining $l^{j..k}$ with $l^{j..j}$. If the element description of l is completely removed, then we define $e(l)$ as `empty`. We observe that, given a label l, there is a single document X_{min} that satisfies D^{min} and is rooted at l, if D is disjunction-free. This document will be used in the generation process when we must satisfy a count constraint with $\leq$. (If X_{min} does not satisfy such a count constraint, then no satisfying document exists.)

Example 6.3. Consider the DTD D from Example 1.1. The minimal version of D is the DTD

$$\mathsf{state} \leftarrow \mathsf{name}, \mathsf{city} \qquad \mathsf{store} \leftarrow \mathsf{address}, \mathsf{product}^{50..50}$$
$$\mathsf{city} \leftarrow \mathsf{name}, \mathsf{store} \qquad \mathsf{product} \leftarrow \mathsf{name}, \mathsf{price}$$

To satisfy the count constraint $\mathsf{count}(\Downarrow[\mathsf{discount}]) \leq 0$ from Figure 2, we can simply create the single document rooted at store that satisfies the minimal DTD. □

6.2 Horizontal Axes

Another extension of interest is to allow horizontal axes in the XPath expressions. We consider two types of horizontal axes: $\rightarrow$ and $\Rightarrow$, which are the immediate-sibling and following-sibling-or-self axes, respectively.

For $\rightarrow$, determining satisfiability is intractable, both for count constraints with $\geq$ and with $\leq$.

Theorem 6.4. *The satisfiability problem is NP-hard for disjunction-free nonrecursive DTDs and legal GxBE commands G, if G contains count constraints using the axis $\rightarrow$.*

If G contains only $\Rightarrow$, then it is possible to generate a document satisfying G by an adaptation of GenConSat.[7] This adaptation requires a new implementation of $\mathrm{NFC}(l, P)$ to correctly determine the XPath expressions that must be satisfied, based on the sibling relationships appearing in P.

6.3 Bounded-Size Satisfiability

A final variation of interest is *bounded-size satisfiability*, i.e., determining satisfiability of D and G when there is a bound on the size of the document that should be created.

Problem 3 (**(θ, k)-Bounded Satisfiability**). *Let D be a DTD, G be a GxBE command, k be an integer and $\theta \in \{\leq, \geq\}$. Determine whether there exists a document X such that $X \models (G, D)$ and $|X|\,\theta\, k$ (where $|X|$ is the number of nodes in X).*

$(\leq, k)$-Bounded satisfiability is useful to obtain a small output document, which may be helpful during the testing process since it is likely to be easily understood. On the other hand, $(\geq, k)$-bounded satisfiability is of particular interest when considering constraints that use the $\leq$ operator, since it can determine the existence of large documents satisfying the user constraints. Unfortunately, the following theorem shows that the interesting cases are intractable.

Theorem 6.5. *For GxBE commands containing only $\geq$ and disjunction-free nonrecursive DTDs, $(\leq, k)$-bounded satisfiability is NP-hard and $(\geq, k)$-bounded satisfiability is polynomial.*

For GxBE commands containing only $\leq$ and disjunction-free nonrecursive DTDs, $(\leq, k)$-bounded satisfiability is polynomial and $(\geq, k)$-bounded satisfiability is NP-hard.

7. CONCLUSION

We presented a natural and general framework for automatically generating XML documents, that satisfy a DTD as well as global properties. Our generation algorithm is general enough to create any document satisfying a given GxBE command, and can return many different documents, if the nondeterministic sub-procedures (e.g., GenArbitraryDoc) return different values when they are reevaluated. We also studied several important extensions of the language of GxBE commands and identified both tractable and intractable cases.

Thus far, the documents created contain only structure, and no data (aside from data explicitly written in the GxBE document). As future work, we intend to study adding data values (including ID/IDREF values) which satisfy an expected data distribution. We believe that this can be done in the spirit of [2], by adding meta-data to the DTD label definitions. We also plan to implement our algorithms.

Our procedures can produce many documents satisfying the user constraints (Remark 4). Another interesting problem is to create a random document satisfying the user constraints. Due to the relationship between counting and generation [9], it is likely that generating a random document will be possible only if we have the ability to count the number of distinct documents satisfying a given constraint. This is an interesting problem for future work.

Finally, we intend to extend our language of DTDs to allow arbitrary subexpressions (and not only labels) to have a minimal and maximal number of occurrences. This involves some intricacies, e.g., in defining the Glushkov automatons and in checking that the DTD is 1-unambiguous [10].

[7]A syntactic requirement that $\Rightarrow$ only appears after at least one $\downarrow$ is needed to ensure that our constraints only constrain data that is within a specifically defined context node.

8. REFERENCES

[1] A. Aboulnaga, J. Naughton, and C. Zhang. Generating synthetic complex-structured XML data. In *WebDB*, 2001.

[2] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: an extensible template-based data generator for XML. In *WebDB*, 2002.

[3] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of dtds. In *PODS*, 2005.

[4] C. Binnig, D. Kossman, and E. Lo. Testing database applications. In *SIGMOD*, 2006.

[5] N. Bruno and S. Chaudhuri. Flexible database generators. In *VLDB*, 2005.

[6] P. Caron and D. Ziadi. Characterization of glushkov automata. *Theoretical Computer Science*, 233:75–90, 2000.

[7] S. Cohen. Count-constraints for generating XML. In *NGITS*, Kibbutz Shefayim, Israel, 2006.

[8] K. Houkjaer, K. Torp, and R. Wind. Simple and realistic data generation. In *VLDB*, 2006.

[9] M. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43:169–188, 1986.

[10] P. Kilpeläinen and R. Tuhkanen. One-unambiguity of regular expressions with numeric occurrence indicators. *Inf. Comput.*, 205(6):890–916, 2007.

[11] A. Neufeld, G. Moerkotte, and P. C. Lockemann. Generating consistent test data for a variable set of general consistency constraints. *The VLDB Journal*, 2(2):173–213, 1993.

[12] K. Runapongsa, J. Patel, H. Jagadish, Y. Chen, and S. Al-Khalifa. The Michigan benchmark: towards XML query performance diagnostics. *Information Systems*, 31(2):73–97, 206.

[13] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: a benchmark for XML data management. In *VLDB*, 2002.

[14] M. Y. Vardi. The complexity of relational query languages. In *STOC*, 1982.

[15] B. Yao, M. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *ICDE*, 2004.