A Normal Form for XML Documents

Marcelo Arenas
Department of Computer Science
University of Toronto
marenas@cs.toronto.edu

Leonid Libkin*
Department of Computer Science
University of Toronto
libkin@cs.toronto.edu

Abstract

This paper takes a first step towards the design and normalization theory for XML documents. We show that, like relational databases, XML documents may contain redundant information, and may be prone to update anomalies. Furthermore, such problems are caused by certain functional dependencies among paths in the document. Our goal is to find a way of converting an arbitrary DTD into a well-designed one, that avoids these problems. We first introduce the concept of a functional dependency for XML, and define its semantics via a relational representation of XML. We then define an XML normal form, XNF, that avoids update anomalies and redundancies. We study its properties and show that it generalizes BCNF and a normal form for nested relations when those are appropriately coded as XML documents. Finally, we present a lossless algorithm for converting any DTD into one in XNF.

1 Introduction

The concepts of database design and normal forms are a key component of the relational database technology. In this paper, we study design principles for XML data. XML has recently emerged as a new basic format for data exchange. Although many XML documents are views of relational data, the number of applications using native XML documents is increasing rapidly. Such applications may use native XML storage facilities [20], and update XML data [28]. Updates, like in relational databases, may cause anomalies if data is redundant. In the relational world, anomalies are avoided by using well-designed database schema. XML has its version of schema too; most often it is DTDs (Document Type Definitions), and some other proposals exist or are under development [31, 30]. What would it mean then for such a schema to be well or poorly designed? Clearly, this question has arisen in practice: one can find companies offering help in "good DTD design." This help, however, comes in form of consulting services rather than commercially available software, as there are no clear guidelines for producing well designed XML.

Our goal is to find principles for good XML data design, and algorithms to produce such designs. We believe that it is important to do this research now, as a lot of data is being put on the web. Once massive web databases are created, it is very hard to change their organization; thus, there is a risk of having large amounts of widely accessible, but at the same time poorly organized legacy data.

Normalization is one of the most thoroughly researched subjects in database theory (a survey [4] produced many references more than 20 years ago), and cannot be reconstructed in a single paper in its entirety. Here we follow the standard treatment of one of the most common (if not the most common) normal forms, BCNF. It eliminates redundancies and avoids update anomalies which they cause by decomposing into relational subschemas in which every nontrivial functional dependency defines a key. Just to retrace this development in the XML context, we need the following:

a) Understanding of what a redundancy and an update anomaly is.

b) A definition and basic properties of functional dependencies (so far, most proposals for XML constraints concentrate on keys).

c) A definition of what "bad" functional dependencies are (those that cause redundancies and update anomalies).

d) An algorithm for converting an arbitrary DTD into one that does not admit such bad functional dependencies.

*Research affiliation: Bell Laboratories.

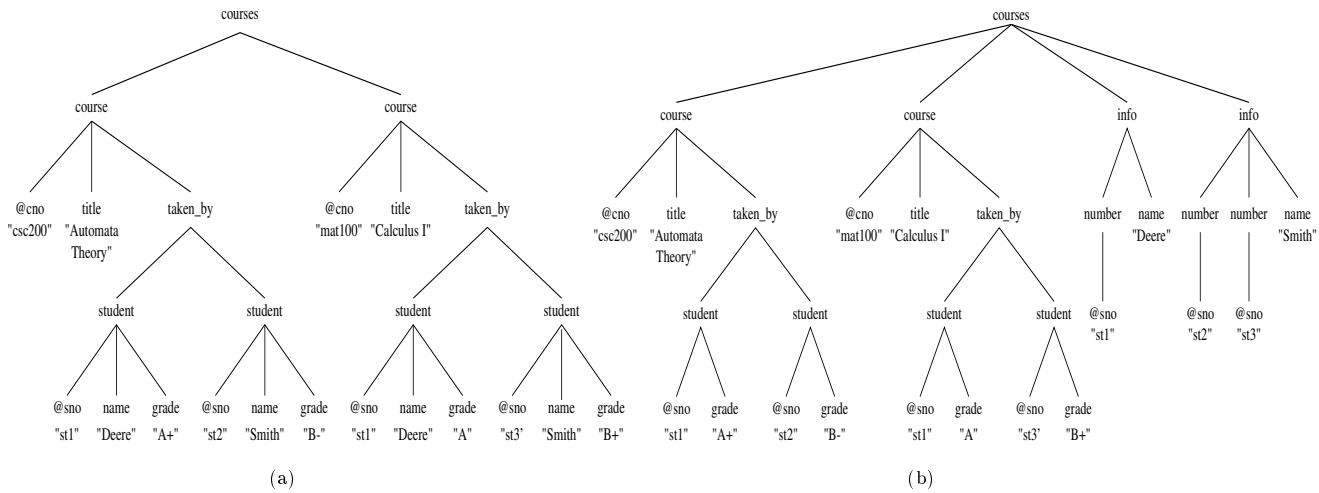


Figure 1: Examples of XML documents.

Starting with point a), how does one identify bad designs? We have looked at a large number of DTDs and found two kinds of commonly present design problems. They are illustrated in two examples below.

Example 1.1: Consider the following DTD that describes a part of a university database:

```
<!ELEMENT courses (course*)>
<!ELEMENT course (title, taken_by)>
  <!ATTLIST course
            cno CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT taken_by (student*)>
<!ELEMENT student (name, grade)>
  <!ATTLIST student
            sno CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT grade (#PCDATA)>
```

For every course, we store its number (`cno`), its title and the list of students taking the course. For each student taking a course, we store his/her number (`sno`), name, and the grade in the course.

An example of an XML document that conforms to this DTD is shown in Figure 1, (a). This document satisfies the following constraint: any two `student` elements with the same `sno` value must have the same `name`. This constraint (which looks very much like a functional dependency), causes the document to store redundant information: for example, the name `Deere` for student `st1` is stored twice. And just as in relational databases, such redundancies can lead to update anomalies: for example, updating the name of `st1` for only one course results in an inconsistent document, and removing the student from a course may result in removing that student from the document altogether.

In order to eliminate redundant information, we use a technique similar to the relational one, and split the information about the name and the grade. Since we deal with just one XML document, we must do it by creating an extra element type, `info`, for student information, as shown below:

```
<!ELEMENT courses (course*, info*)>
<!ELEMENT course (title,taken_by)>
  <!ATTLIST course
            cno CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT taken_by (student*)>
<!ELEMENT student (grade)>
  <!ATTLIST student
            sno CDATA #REQUIRED>
<!ELEMENT grade (#PCDATA)>
<!ELEMENT info (number*,name)>
<!ELEMENT number EMPTY>
  <!ATTLIST number
            sno CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
```

Each `info` element has as children one `name` and a sequence of `number` elements, with `sno` as an attribute. Different students can have the same name, and we group all student numbers `sno` for each `name` under the same `info` element. A restructured document that conforms to this DTD is shown in Figure 1, (b). Note that `st2` and `st3` are put together because both students have the same name.

This example is reminiscent of the canonical example of bad relational design caused by non-key functional dependencies, and so is the modification of the schema. Some examples of redundancies are more closely related to the hierarchical structure of XML documents.

Example 1.2: The DTD below is a part of the DBLP database [8] for storing data about conferences.

```
<!ELEMENT db (conf*)>
<!ELEMENT conf (title, issue+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT issue (inproceedings+)>
<!ELEMENT inproceedings (author+,
            title, booktitle)>
  <!ATTLIST inproceedings
            key ID #REQUIRED
            pages CDATA #REQUIRED
            year CDATA #REQUIRED>
<!ELEMENT author (#PCDATA)>
<!ELEMENT booktitle (#PCDATA)>
```

Each conference has a title, and one or more issues (which correspond to years when the conference was held). Papers are stored in `inproceedings` elements; the year of publication is one of its attributes.

Such a document satisfies the following constraint: any two `inproceedings` children of the same `issue` must have the same value of `year`. This too is similar to relational functional dependencies, but now we refer to the values (the `year` attribute) as well as the structure (children of the same `issue`). Moreover, we only talk about `inproceedings` nodes that are children of the same `issue` element. Thus, this functional dependency can be considered relative to each `issue`.

The functional dependency here leads to redundancy: year is stored multiple times for a conference. The natural solution to the problem in this case is not to create a new element for storing the year, but rather restructure the document and make `year` an attribute of `issue`. That is, we change attribute lists as:

```
<!ATTLIST issue
            year CDATA #REQUIRED>
<!ATTLIST inproceedings
            key ID #REQUIRED
            pages CDATA #REQUIRED>
```

Our goal is to show how to detect anomalies of those kinds, and to transform documents in a lossless fashion into ones that do not suffer from those problems.

The first step towards that goal is to introduce functional dependencies (FDs) for XML documents. So far, most proposals for XML constraints deal with keys and foreign keys [5, 6, 31]. We introduce FDs for XML by considering a relational representation of documents and defining FDs on them. The relational representation is somewhat similar to the total unnesting of a nested relation [26, 29]; however, we have to deal with DTDs that may contain arbitrary regular expressions, and be recursive. Our representation via *tree tuples*, introduced in Section 3, may contain null values. In Section 4, XML FDs are introduced via FDs on incomplete relations [3, 21].

The next step is the definition of a normal form that disallows redundancy-causing FDs. We give it in Section 5, and show that our normal form, called XNF, generalizes BCNF and a nested normal form NNF [22, 23].

The last step then is to find an algorithm that converts any DTD into one in XNF. We do this in Section 6. On both examples shown earlier, the algorithm produces exactly the desired reconstruction of the DTD. The main algorithm uses implication of functional dependencies (although there is a version that does not use implication, but it may produce suboptimal results). In Section 7, we show that for a large class of DTDs, covering most DTDs that occur in practice, the implication problem is tractable (in fact, quadratic).

One of the reasons for the success of the normalization theory is its simplicity, at least for the commonly used normal forms such as BCNF, 3NF and 4NF. Hence, the normalization theory for XML should not be extremely complicated in order to be applicable. In particular, this was the reason we chose to use DTDs instead of more complex formalisms [31, 30]. This is in perfect analogy with the situation in the relational world: although SQL DDL is a rather complicated language with numerous features, BCNF decomposition uses a simple model of a set of attributes and a set of functional dependencies.

Related work For survey of relational normalization, see [1, 4]. Normalization for nested relations and object-oriented databases is studied in [23, 22, 27]. Coding nested relations into flat ones, similar to our tree tuples, is done in [26, 29]. We use FDs and relational algebra queries over incomplete relations using the techniques from [3, 7, 15, 19, 21]. XML constraints (mostly keys) have been studied in [5, 6, 12]; these constraints do not use DTDs. XML constraints that takes DTDs into account are studied in [11]. Finally, [2] considers normal forms for extended context-free grammars similar to the Greibach normal form for CFGs; these, however, do not necessarily guarantee good XML design.

2 Notations

Assume that we have the following disjoint sets: *El* of element names, *Att* of attribute names, *Str* of possible values of string-valued attributes, and *Vert* of node identifiers. All attribute names start with the symbol @, and these are the only ones starting with this symbol. We let S and $\perp$ (null) be reserved symbols not in any of those sets.

Definition 1 *A* DTD *(Document Type Definition) is defined to be* $D = (E, A, P, R, r)$*, where:*

- $E \subseteq El$ *is a finite set of* element types.
- $A \subseteq Att$ *is a finite set of* attributes.

- *P is a mapping from E to* element type definitions*: Given $\tau \in E$, $P(\tau) = \mathtt{S}$ or $P(\tau)$ is a regular expression α defined as follows:*

$$\alpha \quad ::= \quad \epsilon \mid \tau' \mid \alpha|\alpha \mid \alpha,\alpha \mid \alpha^*$$

  where ϵ is the empty sequence, $\tau' \in E$, and "$|$", "$,$" and "$*$" denote union, concatenation, and the Kleene closure, respectively.

- *R is a mapping from E to the powerset of A. If $@l \in R(\tau)$, we say that $@l$ is* defined for *τ.*

- *$r \in E$ and is called* the element type of the root*. Without loss of generality, we assume that r does not occur in $P(\tau)$ for any $\tau \in E$.*

The symbols ϵ and `S` represent element type declarations `EMPTY` and `#PCDATA`, respectively.

Given a DTD $D = (E, A, P, R, r)$, a string $w = w_1 \cdots w_n$ is a *path* in D if $w_1 = r$, w_i is in the alphabet of $P(w_{i-1})$, for each $i \in [2, n-1]$, and w_n is in the alphabet of $P(w_{n-1})$ or $w_n = @l$ for some $@l \in R(w_{n-1})$. We define $length(w)$ as n and $last(w)$ as w_n. We let $paths(D)$ stand for the set of all paths in D and $EPaths(D)$ for the set of all paths that ends with an element type (rather than an attribute or `S`); that is, $EPaths(D) = \{p \in paths(D) \mid last(p) \in E\}$. A DTD is called *recursive* if $paths(D)$ is infinite.

Definition 2 *An* XML tree *T is defined to be a tree $(V, lab, ele, att, root)$, where*

- *$V \subseteq Vert$ is a finite set of* vertices *(*nodes*).*

- *$lab : V \to El$.*

- *$ele : V \to Str \cup V^*$.*

- *att is a partial function $V \times Att \to Str$. For each $v \in V$, the set $\{@l \in Att \mid att(v, @l)$ is defined$\}$ is required to be finite.*

- *$root \in V$ is called* the root of *T.*

The parent-child edge relation on V, $\{(v_1, v_2) \mid v_2$ occurs in $ele(v_1)\}$, is required to form a rooted tree.

Notice that we do not allow mixed content in XML trees. The children of an element node can be either zero or more element nodes or one string.

Given an XML tree T, a string $w_1 \cdots w_n$, with $w_1, \ldots, w_{n-1} \in El$ and $w_n \in El \cup Att \cup \{\mathtt{S}\}$, is a *path* in T if there are vertices $v_1 \cdots v_{n-1}$ in V such that:

- $v_1 = root$, v_{i+1} is a child of v_i $(1 \leq i \leq n-2)$, $lab(v_i) = w_i$ $(1 \leq i \leq n-1)$.

- If $w_n \in El$, then there is a child v_n of v_{n-1} such that $lab(v_n) = w_n$. If $w_n = @l$, with $@l \in Att$, then $att(v_{n-1}, @l)$ is defined. If $w_n = \mathtt{S}$, then v_{n-1} has a child in Str.

We let $paths(T)$ stand for the set of paths in T.

We next give a standard definition of a tree conforming to a DTD ($T \models D$) as well as a weaker version of T being compatible with D ($T \lhd D$).

Definition 3 *Given a DTD $D = (E, A, P, R, r)$ and an XML tree $T = (V, lab, ele, att, root)$, we say that T* conforms to *D ($T \models D$) if*

- *lab is a mapping from V to E.*

- *For each $v \in V$, if $P(lab(v)) = \mathtt{S}$, then $ele(v) = [s]$, where $s \in Str$. Otherwise, if $ele(v) = [v_1, \ldots, v_n]$, then the string $lab(v_1) \cdots lab(v_n)$ must be in the regular language defined by $P(lab(v))$.*

- *att is a partial function from $V \times A$ to Str such that for any $v \in V$ and $@l \in A$, $att(v, @l)$ is defined iff $@l \in R(lab(v))$.*

- *$lab(root) = r$.*

We say that T is compatible with *D (written $T \lhd D$) iff $paths(T) \subseteq paths(D)$.*

3 Tree Tuples

To extend the notions of functional dependencies to the XML setting, we represent XML trees as sets of tuples. While various mappings from XML to the relational model have been proposed [14, 25], the mapping that we use is of a different nature, as our goal is not to find a way of storing documents efficiently, but rather find a correspondence between documents and relations that lends itself to a natural definition of functional dependency.

Various languages proposed for expressing XML integrity constraints such as keys, [5, 6, 31], treat XML trees as unordered (for the purpose of defining the semantics of constraints): that is, the order of children of any given node is irrelevant as far as satisfaction of constraints is concerned. In XML trees, on the other hand, children of each node are ordered. We first define a notion of subsumption that disregard this ordering.

Given two XML trees $T_1 = (V_1, lab_1, ele_1, att_1, root_1)$ and $T_2 = (V_2, lab_2, ele_2, att_2, root_2)$, we say that T_1 is subsumed by T_2, written as $T_1 \preceq T_2$ if

- $V_1 \subseteq V_2$.

- $root_1 = root_2$.

- $lab_2{\restriction_{V_1}} = lab_1$.
- $att_2{\restriction_{V_1 \times Att}} = att_1$.
- For all $v \in V_1$, $ele_1(v)$ is a sublist of a permutation of $ele_2(v)$.

This relation is a pre-order, which gives rise to an equivalence relation: $T_1 \equiv T_2$ iff $T_1 \preceq T_2$ and $T_2 \preceq T_1$. That is, $T_1 \equiv T_2$ iff T_1 and T_2 are equal as unordered trees. We define $[T]$ to be the $\equiv$-equivalence class of T.

We write $[T] \models D$ if $T_1 \models D$ for some $T_1 \in [T]$. It is easy to see that for any $T_1 \equiv T_2$, $paths(T_1) = paths(T_2)$; hence $T_1 \lhd D$ iff $T_2 \lhd D$. We shall also write $T_1 \prec T_2$ when $T_1 \preceq T_2$ and $T_2 \not\preceq T_1$.

Definition 4 (Tree tuples) *Given a DTD $D = (E, A, P, R, r)$, a* tree tuple *t in D is a function from $paths(D)$ to $Vert \cup Str \cup \{\perp\}$ such that:*

- *For $p \in EPaths(D)$, $t(p) \in Vert \cup \{\perp\}$, and $t(r) \neq \perp$.*
- *For $p \in paths(D) - EPaths(D)$, $t(p) \in Str \cup \{\perp\}$.*
- *If $t(p_1) = t(p_2)$ and $t(p_1) \in Vert$, then $p_1 = p_2$.*
- *If $t(p_1) = \perp$ and p_1 is a prefix of p_2, then $t(p_2) = \perp$.*
- *$\{p \in paths(D) \mid t(p) \neq \perp\}$ is finite.*

$\mathcal{T}(D)$ is defined to be the set of all tree tuples in D. For a tree tuple t and a path p, we write $t.p$ for $t(p)$.

Example 3.1: Suppose that D is the DTD shown in example 1.1 (a). Then a tree tuple in D assigns values to each path in $paths(D)$ as is shown in figure 2 (a).

We use nulls ($\perp$) in tree tuples because of the disjunction in DTDs. For example, let $D = (E, A, P, R, r)$, where $E = \{r, a, b\}$, $A = \emptyset$, $P(r) = (a|b)$, $P(a) = \epsilon$ and $P(b) = \epsilon$. Then $paths(D) = \{r, r.a, r.b\}$ but no tree tuple coming from an XML tree conforming to D can assign non-null values to both $r.a$ and $r.b$.

If D is a recursive DTD, then $paths(D)$ is infinite; however, only a finite number of values in a tree tuple are different from $\perp$. For each tree tuple t, its non-null values give rise to an XML tree as follows.

Definition 5 ($tree_D$) *Given a DTD $D = (E, A, P, R, r)$ and a tree tuple $t \in \mathcal{T}(D)$, $tree_D(t)$ is defined to be an XML tree $(V, lab, ele, att, root)$, where $root = t.r$ and*

- *$V = \{v \in Vert \mid \exists p \in paths(D)$ such that $v = t.p\}$.*
- *If $v = t.p$, then $lab(v) = last(p)$.*
- *If $v = t.p$, then $ele(v)$ is defined to be the list containing $\{t.p' \mid t.p' \neq \perp$ and $p' = p.\tau, \tau \in E$, or $p' = p.\mathtt{S}\}$, ordered lexicographically.*
- *If $v = t.p$, $@l \in A$ and $t.p.@l \neq \perp$, then $att(v, @l) = t.p.@l$.*

Example 3.2: The non-null values of the tree tuple t shown in figure 2 (a) give rise to the XML tree shown in figure 2 (b).

Note that $tree_D(t)$ need not conform to the DTD D, but:

Proposition 1 *If $t \in \mathcal{T}(D)$, then $tree_D(t) \lhd D$.* □

We would like to describe XML trees in terms of the tuples they contain. For this, we need to select tuples containing the maximal amount of information. This is done via the usual notion of ordering on tuples (and relations) with nulls, [7, 15, 16]. If we have two tree tuples t_1, t_2, we write $t_1 \sqsubseteq t_2$ if whenever $t_1.p$ is defined, then so is $t_2.p$, and $t_1.p \neq \perp$ implies $t_1.p = t_2.p$. As usual, $t_1 \sqsubset t_2$ means $t_1 \sqsubseteq t_2$ and $t_1 \neq t_2$. Given two sets of tree tuples, X and Y, we write $X \sqsubseteq^\flat Y$ if $\forall t_1 \in X \exists t_2 \in Y$ $t_1 \sqsubseteq t_2$.

Definition 6 ($tuples_D$) *Given a DTD D and an XML tree T such that $T \lhd D$, $tuples_D(T)$ is defined to be the set of maximal, wrt $\sqsubseteq$, tree tuples t such that $tree_D(t)$ is subsumed by T; that is:*

$$\max_{\sqsubseteq}\{t \in \mathcal{T}(D) \mid tree_D(t) \preceq T\}.$$

Observe that $T_1 \equiv T_2$ implies $tuples_D(T_1) = tuples_D(T_2)$. Hence, $tuples_D$ applies to equivalence classes: $tuples_D([T]) = tuples_D(T)$.

Proposition 2 *If $T \lhd D$, then $tuples_D(T)$ is a finite subset of $\mathcal{T}(D)$. Furthermore, $tuples_D(\cdot)$ is monotone: $T_1 \preceq T_2$ implies $tuples_D(T_1) \sqsubseteq^\flat tuples_D(T_2)$.* □

Finally, we define the trees represented by a set of tuples X as the minimal, with respect to $\preceq$, trees containing all tuples in X.

Definition 7 ($trees_D$) *Given a DTD D and a set of tree tuples $X \subseteq \mathcal{T}(D)$, $trees_D(X)$ is defined to be:*

$$\min_{\preceq}\{T \mid T \lhd D \text{ and } \forall t \in X,\ tree_D(t) \preceq T\}.$$

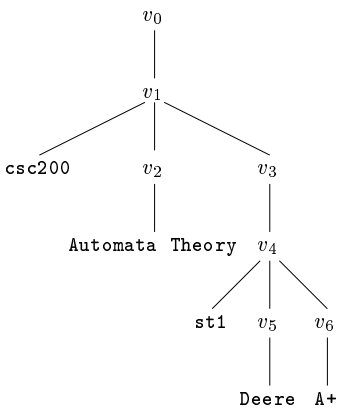
For $T \in trees_D(X)$ and $T' \equiv T$, $T' \in trees_D(X)$; thus $trees_D(X)$ is a union of $\equiv$ equivalence classes.

The following shows that every XML document can be represented as a set of tree tuples, if we consider it as an unordered tree. That is, a tree T can be reconstructed from $tuples_D(T)$, up to equivalence $\equiv$.

Theorem 1 *Given a DTD D and an XML tree T, if $T \lhd D$, then $trees_D(tuples_D([T])) = [T]$.* □

$t(courses) = v_0$
$t(courses.course) = v_1$
$t(courses.course.@cno) = \texttt{csc200}$
$t(courses.course.title) = v_2$
$t(courses.course.title.\texttt{S}) = \texttt{Automata Theory}$
$t(courses.course.taken_by) = v_3$
$t(courses.course.taken_by.student) = v_4$
$t(courses.course.taken_by.student.@sno) = \texttt{st1}$
$t(courses.course.taken_by.student.name) = v_5$
$t(courses.course.taken_by.student.name.\texttt{S}) = \texttt{Deere}$
$t(courses.course.taken_by.student.grade) = v_6$
$t(courses.course.taken_by.student.grade.\texttt{S}) = \texttt{A+}$

(a) Values of t



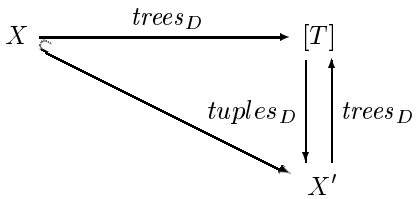
(b) $tree_D(t)$

Figure 2: Tree tuple t and its tree representation.

The converse does not hold, but can be partially recovered when $trees_D(X)$ is a single equivalence class. We say that $X \subseteq \mathcal{T}(D)$ is *D-compatible* if there is an XML tree T such that $T \lhd D$ and $X \subseteq tuples_D(T)$.

Proposition 3 *If $X \subseteq \mathcal{T}(D)$ is D-compatible, then (a) There is an XML tree T such that $T \lhd D$ and $trees_D(X) = [T]$, and (b) $X \sqsubseteq^\flat tuples_D(trees_D(X))$.*

Theorem 1 and Proposition 3 are summarized in the diagram presented in the following figure. In this diagram, X is a D-compatible set of tree tuples. The arrow $\subset\!\!\longrightarrow$ stands for the $\sqsubseteq^\flat$ ordering.



4 Functional Dependencies

We define functional dependencies for XML by using tree tuples. For a DTD D, a *functional dependency (FD)* over D is an expression of the form $S_1 \rightarrow S_2$ where S_1, S_2 are finite non-empty subsets of $paths(D)$. The set of all FDs over D is denoted by $\mathcal{FD}(D)$.

For $S \subseteq paths(D)$, and $t, t' \in \mathcal{T}(D)$, $t.S = t'.S$ means $t.p = t'.p$ for all $p \in S$. Furthermore, $t.S \neq \perp$ means $t.p \neq \perp$ for all $p \in S$.

If $S_1 \rightarrow S_2 \in \mathcal{FD}(D)$ and T is an XML tree such that $T \lhd D$ and $S_1 \cup S_2 \subseteq paths(T)$, we say that T *satisfies* $S_1 \rightarrow S_2$ (written $T \models S_1 \rightarrow S_2$) if for every $t_1, t_2 \in tuples_D(T)$, $t_1.S_1 = t_2.S_1$ and $t_1.S_1 \neq \perp$ imply $t_1.S_2 = t_2.S_2$. This extends to equivalence classes, since for any FD φ, and $T \equiv T'$, $T \models \varphi$ iff $T' \models \varphi$.

We write $T \models \Sigma$, for $\Sigma \subseteq \mathcal{FD}(D)$, if $T \models \varphi$ for each $\varphi \in \Sigma$, and we write $T \models (D, \Sigma)$, if $T \models D$ and $T \models \Sigma$.

Example 4.1: Referring back to Example 1.1, we have the following FDs. `cno` is a key of `course`:

$$courses.course.@cno \rightarrow courses.course. \quad \text{(FD1)}$$

Another FD says that two distinct `student` subelements of the same `course` cannot have the same `sno`:

$$\{\ courses.course,\ courses.course.taken_by.student.@sno\ \} \rightarrow courses.course.taken_by.student. \quad \text{(FD2)}$$

Finally, to say that two `student` elements with the same `sno` value must have the same `name`, we use

$$courses.course.taken_by.student.@sno \rightarrow courses.course.taken_by.student.name.\texttt{S}. \quad \text{(FD3)}$$

We offer a few remarks on our definition of FDs. First, using the tree tuples representation, it is easy to combine node and value equality: the former corresponds to equality between vertices and the latter to equality between strings. Moreover, keys naturally appear as a subclass of FDs, and relative constraints can also be encoded. Note that by defining the semantics of $\mathcal{FD}(D)$ on $\mathcal{T}(D)$, we essentially define satisfaction of FDs on relations with null values, and our semantics is the standard semantics used in [3, 21].

Given a DTD D, a set $\Sigma \subseteq \mathcal{FD}(D)$ and $\varphi \in \mathcal{FD}(D)$, we say that (D, Σ) *implies* φ, written $(D, \Sigma) \vdash \varphi$, if for any tree T with $T \models D$ and $T \models \Sigma$, it is the case that $T \models \varphi$. The set of all FDs implied by (D, Σ) will be denoted by $(D, \Sigma)^+$.

An FD φ is *trivial* if $(D, \emptyset) \vdash \varphi$. In relational databases, the only trivial FDs are $X \to Y$, with $Y \subseteq X$. Here, DTD forces some more interesting trivial FDs. For instance, for each $p \in EPaths(D)$ and p' a prefix of p, $(D, \emptyset) \vdash p \to p'$. Furthermore, for $p, p.@l \in paths(D)$, $(D, \emptyset) \vdash p \to p.@l$.

5 XNF: An XML Normal Form

With the definitions of the previous section, we are ready to present the normal form that generalizes BCNF for XML documents.

Definition 8 *Given a DTD D and $\Sigma \subseteq \mathcal{FD}(D)$, (D, Σ) is in* XML normal form (XNF) *iff for every non-trivial FD $\varphi \in (D, \Sigma)^+$ of the form $S \to p.@l$ or $S \to p.\mathtt{S}$, it is the case that $S \to p$ is in $(D, \Sigma)^+$.*

The intuition is as follows. Suppose that $S \to p.@l$ is in $(D, \Sigma)^+$. If T is an XML tree conforming to D and satisfying Σ, then in T for every set of values of the elements in S, we can find only one value of $p.@l$. Thus, for every set of values of S we need to store the value of $p.@l$ only once; in other words, $S \to p$ must be implied by (D, Σ).

In this definition, we impose the condition that φ is a non-trivial FD. Indeed, the trivial FD $p.@l \to p.@l$ is always in $(D, \Sigma)^+$, but often $p.@l \to p \notin (D, \Sigma)^+$, which does not necessarily represent a bad design.

To show how XNF distinguishes good XML design from bad design, we revisit the examples from the introduction, and prove that XNF generalizes BCNF and NNF, a normal form for nested relations [22, 23].

Example 5.1: Consider the DTD from example 1.1 (a) whose FDs are (FD1), (FD2), (FD3) shown in the previous section. (FD3) associates a unique name with each student number, which is therefore redundant. The design is *not* in XNF, since it contains (FD3) but does not imply the FD

$$courses.course.taken_by.student.@sno \to courses.course.taken_by.student.name$$

To remedy this, we gave a revised DTD in example 1.1 (b). The idea was to create a new element `info` for storing information about student. That design satisfies FDs (FD1), (FD2) as well as

$$courses.info.number.@sno \to courses.info,$$

and can be easily verified to be in XNF.

Example 5.2: Suppose that D is the DBLP DTD from example 1.2. Among the set Σ of FDs satisfied by the documents are:

$$db.conf.title.\mathtt{S} \to db.conf \quad \text{(FD4)}$$

$$db.conf.issue \to db.conf.issue.inproceedings.@year \quad \text{(FD5)}$$

For each issue of a conference, its year is stored in every article in that issue; thus, (D, Σ) is not in XNF, since

$$db.conf.issue \to db.conf.issue.inproceedings$$

is not in $(D, \Sigma)^+$.

The solution we proposed in the introduction was to make `year` an attribute of `issue`. (FD5) is not valid in the revised specification, which can be easily verified to be in XNF. Note that we do not replace (FD5) by $db.conf.issue \to db.conf.issue.@year$, since it is a trivial FD and thus is implied by the new DTD alone.

BCNF and XNF Relational databases can be easily mapped into XML documents. Given a schema $G(A_1, \ldots, A_n)$, a DTD D_G has two element types db and G, $P(db) = G^*$, $P(G) = \epsilon$, and $R(G) = \{@A_1, \ldots, @A_n\}$. For a set F of FDs over G, we define a set Σ_F of FDs over D_G that includes, for each $A_{i_1} \cdots A_{i_m} \to A_j$ in F an FD $\{db.G.@A_{i_1}, \ldots, db.G.@A_{i_m}\} \to db.G.@A_j$, as well as $\{db.G.@A_1, \ldots, db.G.@A_n\} \to db.G$ (to avoid duplicates).

Example 5.3: A schema $G(A, B, C)$ can be coded by the following DTD:

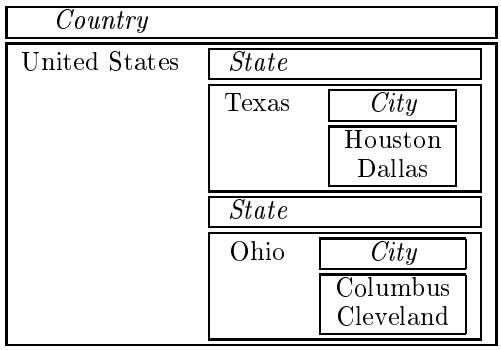
```
<!ELEMENT db (G*)>
<!ELEMENT G EMPTY>
  <!ATTLIST G
            A CDATA #REQUIRED
            B CDATA #REQUIRED
            C CDATA #REQUIRED>
```

In this schema, an FD $A \to B$ is translated into $db.G.@A \to db.G.@B$.

Proposition 4 *(G, F) is in BCNF iff (D_G, Σ_F) is in XNF.* □

NNF and XNF A nested schema is either a set of attributes X, or $X(G_1)^* \ldots (G_n)^*$, where G_i's are nested schemas. An example of a nested relation for the schema $H_1 = Country(H_2)^*$, $H_2 = State(H_3)^*$, $H_3 = City$ is shown in figure 3 (a).

Nested schemas are naturally mapped into DTDs, as they are defined by means of regular expressions. For a



(a) Nested relation H_1

Country	*State*	*City*
United States	Texas	Houston
United States	Texas	Dallas
United States	Ohio	Columbus
United States	Ohio	Cleveland

(b) Complete unnesting of H_1

Figure 3: Nested relation and its unnesting.

nested schema $G = X(G_1)^* \dots (G_n)^*$, we introduce an element type G with $P(G) = G_1^*, \dots, G_n^*$ and $R(G) = \{@A_1, \dots, @A_n\}$, where $X = \{A_1, \dots, A_n\}$; at the top level we have a new element type db with $P(db) = G^*$ and $R(db) = \emptyset$. In our example the DTD is:

```
<!ELEMENT db (H1*)>
<!ELEMENT H1 (H2*)>
  <!ATTLIST H1 Country CDATA #REQUIRED>
<!ELEMENT H2 (H3*)>
  <!ATTLIST H2 State CDATA #REQUIRED>
<!ELEMENT H3 EMPTY>
  <!ATTLIST H3 City CDATA #REQUIRED>
```

The definition of FDs for nested relations uses the notion of complete unnesting. The complete unnesting of a nested relation from our example is shown in figure 3 (b); in general, this notion is easily defined by induction. In our example, we have a valid FD $State \rightarrow Country$, while the FD $State \rightarrow City$ does not hold.

Normalization is usually considered for nested relations in the *partition normal form* (PNF) [1, 22, 23]. A nested relation r over $X(G_1)^* \dots (G_n)^*$ is in PNF if for any two tuples t_1, t_2 in r: (1) if $t_1.X = t_2.X$, then the nested relation $t_1.G_i$ and $t_2.G_i$ are equal, for every $i \in [1, n]$, and (2) each nested relation $t_1.G_i$ must be in PNF, for every $i \in [1, n]$. Note that PNF can be enforced by using FDs on the XML representation. In our example this is done as follows:

$$\begin{aligned} db.H_1.@Country &\rightarrow db.H_1 \\ \{db.H_1,\ db.H_1.H_2.@State\} &\rightarrow db.H_1.H_2 \\ \{db.H_1.H_2,\ db.H_1.H_2.H_3.@City\} &\rightarrow db.H_1.H_2.H_3 \end{aligned}$$

It turns out that one can define FDs over nested relations by using the XML representation. Let U be a set of attributes, G_1 a nested relation schema over U and FD a set of functional dependencies over G_1. Assume that G_1 includes nested relation schemas G_2, ..., G_n and a set of attributes $U' \subseteq U$. For each G_i ($i \in [1, n]$), $path(G_i)$ is inductively defined as follows. If $G_i = G_1$, then $path(G_i) = db.G_1$. Otherwise, if G_i is a nested attribute of G_j, then $path(G_i) = path(G_j).G_i$. Furthermore, if $A \in U'$ is an atomic attribute of G_i, then $path(A) = path(G_i).@A$. For instance, for the schema of the nested relation in Figure 3, $path(H_2) = db.H_1.H_2$ and $path(City) = db.H_1.H_2.H_3.@City$.

We now define Σ_{FD} as follows:

- For each FD $A_{i_1} \cdots A_{i_m} \rightarrow A_i \in FD$, $\{path(A_{i_1}), \dots, path(A_{i_m})\} \rightarrow path(A_i)$ is in Σ_{FD}.
- For each $i \in [1, n]$, if $A_{j_1}, \dots, A_{j_m}$ is the set of atomic attributes of G_i and G_i is a nested attribute of G_j, $\{path(G_j), path(A_{j_1}), \dots, path(A_{j_m})\} \rightarrow path(G_i)$ is in Σ_{FD}.

  Furthermore, if $B_{j_1}, \dots, B_{j_l}$ is the set of atomic attributes of G_1, then $\{path(B_{j_1}), \dots, path(B_{j_l})\} \rightarrow path(G_1)$ is in Σ_{FD}.

Note that the last rule imposes the partition normal form.

A Nested Normal Form (NNF) for nested relations was proposed in [22, 23]. Here we use the presentation of [22] restricted to FDs only. Given a nested relational schema G and a subschema R, for each atomic attribute A of R we define $ancestor(A)$ as the union of the atomic attributes of all the nested relation schemas mentioned in $path(R)$. For instance, $ancestor(State) = \{Country, State\}$. If FD is a set of FDs over G, then say that it is in NNF if for each non-trivial FD $X \rightarrow A$ ($A \in U$), if $X \rightarrow A \in (G, FD)^+$, then $X \rightarrow ancestor(A) \in (G, FD)^+$. As before, $(G, FD)^+$ stands for the set of all FDs implied by (G, FD).

The result below says that a nested relational schema (G, FD) is in NNF if and only if its XML representation,

that consists of a DTD D_G and a set of FDs Σ_{FD} as defined above, is in XNF.

Proposition 5 *(G, FD) is in NNF iff (D_G, Σ_{FD}) is in XNF.* □

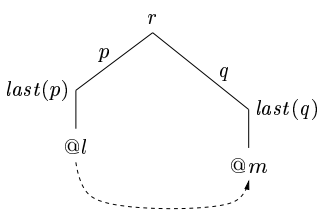
6 Normalizing XML Documents

We show how to transform a DTD D and a set of FDs Σ into a new specification (D', Σ') that is in XNF and contains the same information. Throughout the section, we assume that the DTDs are non-recursive (the recursive case can be handled in a very similar fashion), and that all FDs are of the form: $\{q, p_1.@l_1, \ldots, p_n.@l_n\} \rightarrow p$. That is, they contain at most one element path on the left-hand side. Note that all the FDs we have seen so far are of this form. While constraints of the form $\{q, q', \ldots\}$ are not forbidden, they appear to be quite unnatural, and can be easily eliminated by creating a new attribute $@l$ and splitting $\{q, q'\} \cup S \rightarrow p$ into $q'.@l \rightarrow q'$ and $\{q, q'.@l\} \cup S \rightarrow p$. Furthermore, we assume that paths do not contain the symbol S (since $p.\mathtt{S}$ can always be replaced by a path of the form $p.@l$).

Given a DTD D and a set of FDs Σ, a non-trivial FD $S \rightarrow p.@l$ is called *anomalous*, over (D, Σ), if it violates XNF; that is, $S \rightarrow p.@l \in (D, \Sigma)^+$ but $S \rightarrow p \notin (D, \Sigma)^+$. A path on the right-hand side of an anomalous FD is called an anomalous path, and the set of all such paths is denoted by $AP(D, \Sigma)$.

The algorithm combines two basic ideas presented in the introduction: creating a new element type, and moving an attribute.

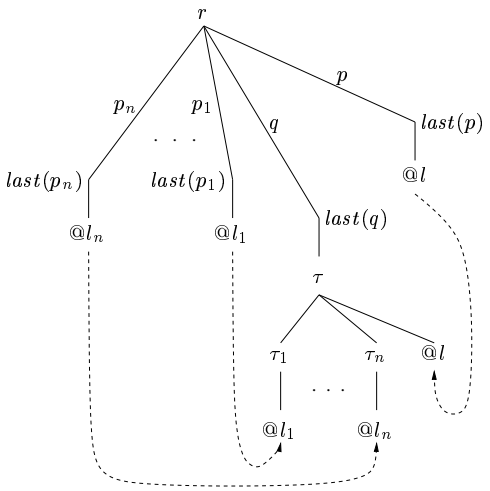
Moving attributes Let $D = (E, A, P, R, r)$ be a DTD, $p.@l \in paths(D)$, $q \in EPaths(D)$ and $@m$ be an attribute. The DTD $D[p.@l := q.@m]$ is constructed by moving the attribute $@l$ from the set of attributes of $last(p)$ to the set of attributes of $last(q)$, and changing its name to $@m$, as shown in the following figure.



Formally, $D[p.@l := q.@m]$ is (E, A', P, R', r), where $A' = A \cup \{@m\}$, $R'(last(q)) = R(last(q)) \cup \{@m\}$, $R'(last(p)) = R(last(p)) \setminus \{@l\}$ and $R'(\tau') = R(\tau')$ for each $\tau' \in E \setminus \{last(q), last(p)\}$. This is the same kind of transformation we saw in moving the `year` attribute in the DBLP example.

Given a set of FDs Σ over D, a set of FDs $\Sigma[p.@l := q.@m]$ over $D[p.@l := q.@m]$ consists of all FDs $S_1 \rightarrow S_2 \in (D, \Sigma)^+$ with $S_1 \cup S_2 \subseteq paths(D[p.@l := q.@m])$.

Creating new element types Let $D = (E, A, P, R, r)$ be a DTD, $S = \{q, p_1.@l_1, \ldots, p_n.@l_n\} \subseteq paths(D)$ such that $n \geq 1$ and $q \in EPaths(D)$. We construct a new DTD D' by creating a new element type τ as a child of the last element of q, making $\tau_1, \ldots, \tau_n$ its children, $@l$ its attribute, and $@l_1, \ldots, @l_n$ attributes of $\tau_1, \ldots, \tau_n$. Furthermore, we remove $@l$ from the set of attributes of the last element of p, as shown in the following figure.



Formally, if $\{\tau, \tau_1, \ldots, \tau_n\}$ are element types which are not in E, the new DTD, denoted by $D[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$, is (E', A, P', R', r), where $E' = E \cup \{\tau, \tau_1, \ldots, \tau_n\}$ and

1. $P'(last(q)) = P(last(q)), \tau^*$, $P'(\tau) = \tau_1^*, \ldots, \tau_n^*$, $P'(\tau_i) = \epsilon$, for each $i \in [1, n]$, and $P'(\tau') = P(\tau')$ for each $\tau' \in E \setminus \{last(q)\}$.
2. $R'(\tau) = \{@l\}$, $R'(\tau_i) = \{@l_i\}$, for each $i \in [1, n]$, $R'(last(p)) = R(last(p)) \setminus \{@l\}$ and $R'(\tau') = R(\tau')$ for each $\tau' \in E \setminus \{last(p)\}$.

Given $D' = D[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$ and a set Σ of FDs over D, we define a set $\Sigma[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$ of FDs over D' as the set that contains the following:

1. $S_1 \rightarrow S_2 \in (D, \Sigma)^+$ with $S_1 \cup S_2 \subseteq paths(D')$;
2. Each FD over q, p_i, $p_i.@l_i$ $(i \in [1, n])$ and $p.@l$ is transferred to τ and its children. That is, if $S_1 \cup S_2 \subseteq \{q, p_1, \ldots, p_n, p_1.@l_1, \ldots, p_n.@l_n, p.@l\}$ and $S_1 \rightarrow S_2 \in (D, \Sigma)^+$, then we include an FD obtained from $S_1 \rightarrow S_2$ by changing p_i to $q.\tau.\tau_i$, $p_i.@l_i$ to $q.\tau.\tau_i.@l_i$, and $p.@l$ to $q.\tau.@l$;

(1) If (D, Σ) is in XNF then return (D, Σ), otherwise go to step (2).

(2) If there is an anomalous FD $S \to p.@l$ with $q \in EPaths(D) \cap S$ and $q \to S \in (D, \Sigma)^+$, then:

$$D := D[p.@l := q.@m]$$
$$\Sigma := \Sigma[p.@l := q.@m]$$

where $@m$ is fresh, and go to step (1).

(3) Choose a (D, Σ)-minimal anomalous FD $S \to p.@l$, where $S = \{q, p_1.@l_1, \ldots, p_n.@l_n\}$. Create fresh element types τ, τ_1, ..., τ_n; set

$$D := D[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$$
$$\Sigma := \Sigma[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$$

and go to step (1).

Figure 4: XNF decomposition algorithm.

3. $\{q,\ q.\tau.\tau_1.@l_1,\ \ldots,\ q.\tau.\tau_n.@l_n\} \to q.\tau$, and $\{q.\tau,\ q.\tau.\tau_i.@l_i\} \to q.\tau.\tau_i$ for $i \in [1, n]$ [1].

This construction, when applied to the student example from the introduction, yields exactly the revised DTD, with τ being `info`, $@l$ being `name`, τ_1 being `number` and $@l_1$ being `sno`.

We are not interested in applying this transformation to an arbitrary anomalous FD, but rather to a *minimal* one. In the relational context, a minimal FD is $X \to A$ such that $X' \not\to A$ for any $X' \subsetneq X$. In our case the definition is a bit more complex to account for paths used in FDs. We say that $\{q, p_1.@l_1, \ldots, p_n.@l_n\} \to p_0.@l_0$ is (D, Σ)-minimal if there is no anomalous FD $S' \to p_i.@l_i \in (D, \Sigma)^+$ such that $i \in [0, n]$ and S' is a subset of $\{q, p_1, \ldots, p_n, p_0.@l_0, \ldots, p_n.@l_n\}$ such that $|S'| \leq n$ and S' contains at most one element path.

Proposition 6 *Let (D', Σ') be constructed from (D, Σ) by using either the "moving attributes" construction, or the "creating new element types" construction applied to a (D, Σ)-minimal FD. Then $AP(D', \Sigma') \subsetneq AP(D, \Sigma)$.* □

The algorithm The algorithm applies the two transformations until the schema is in XNF, as shown in figure 4. It involves FD implication, that is, testing membership in $(D, \Sigma)^+$ (and consequently testing XNF and (D, Σ)-minimality), which will be described in Section 7. Since each step reduces the number of anomalous paths (Proposition 6), we obtain:

Theorem 2 *The XNF decomposition algorithm terminates, and outputs a specification (D, Σ) in XNF.*

[1] If $\perp$ can be a value of $p.@l$ in $tuples_D(T)$, the definition must be modified slightly, by letting $P'(\tau)$ be $\tau_1^*, \ldots, \tau_n^*, (\tau' | \epsilon)$, where τ' is fresh, making $@l$ an attribute of τ', and modifying the definition of FDs accordingly.

Even if testing FD implication is infeasible, one can still decompose into XNF, although the final result may not be as good as with using the implication. A slight modification of the proof of Proposition 6 yields:

Proposition 7 *Consider a simplification of the XNF decomposition algorithm which only consists of step (3) applied to FDs $S \to p.@l \in \Sigma$, and in which the definition of $\Sigma[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$ is modified by using Σ instead of $(D, \Sigma)^+$. Then such an algorithm always terminates and its result is in XNF.*

Lossless Decompositions To prove that our transformations do not lose any information from the documents, we define the concept of lossless decompositions similarly to the relational notion of "generic dominance" from [18]. That notion requires the existence of two relational algebra queries that translate back and forth between two relational schemas. Adapting this definition poses two problems in our setting: first, no XML query language yet has the same "yardstick" status as relational algebra for relational databases, and second, our transformations generate new node ids, which cannot be described by generic queries.

To deal with this, we use the relational representation via the $tuples_D(\cdot)$ operator, and say that (D_2, Σ_2) *is a lossless decomposition of* (D_1, Σ_1), written $(D_1, \Sigma_1) \leq_{\text{lossless}} (D_2, \Sigma_2)$, if there exist relational algebra queries Q_1, Q_1', Q_2 such that for any $T \models (D_1, \Sigma_1)$, there exists $T' \models (D_2, \Sigma_2)$ such that the diagram below commutes:

$$\begin{array}{ccccc} T & & & & T' \\ \downarrow tuples_{D_1} & & & & \downarrow tuples_{D_2} \\ tuples_{D_1}(T) & \underset{Q_1'}{\overset{Q_1}{\rightleftarrows}} & Q_1(tuples_{D_1}(T)) & \xleftarrow{Q_2} & tuples_{D_2}(T') \end{array}$$

The goal of query Q_2 is to eliminate extra node ids that may occur in T' but not in T; then Q_1 and Q_1' go back and forth between $tuples_{D_1}(T)$ and the result of Q_2 on $tuples_{D_2}(T')$. As relations of the form $tuples_D(T)$ may contain nulls, we use the semantics of Codd tables [1, 19] for evaluating relational algebra queries on them.

Proposition 8 *(a) The relation $\leq_{\text{lossless}}$ is transitive. (b) If (D', Σ') is obtained from (D, Σ) by using one of the transformations from the normalization algorithm, then $(D, \Sigma) \leq_{\text{lossless}} (D', \Sigma')$.*

Thus, if (D', Σ') is the output of the normalization algorithm on (D, Σ), then $(D, \Sigma) \leq_{\text{lossless}} (D', \Sigma')$. Moreover, the transformations on the documents can be implemented in XML query languages [13, 32].

7 Reasoning about FDs

In the previous section we saw that it is possible to losslessly convert a DTD into one in XNF. The algorithm used FD implication. We now show that for most classes of DTDs used in practice, this problem is tractable. We assume, without loss of generality, that all FDs have a single path on the right-hand side.

Typically, regular expressions used in DTDs are rather simple. We now formulate a criterion for simplicity that corresponds to the usual practice of writing regular expressions in DTDs. Given an alphabet A, a regular expression over A is called *trivial* if it of the form $s_1, \ldots, s_n$, where for each s_i there is a letter $a_i \in A$ such that s_i is either a_i or $a_i?$ (which abbreviates $a_i|\epsilon$), or a_i^+ or a_i^*, and for $i \neq j$, $a_i \neq a_j$. We call a regular expression s *simple* if there is a trivial regular expression s' such that any word w in the language denoted by s is a permutation of a word in the language denoted by s', and vice versa.

For example, $(a|b|c)^*$ is simple: a^*, b^*, c^* is trivial, and every word in $(a|b|c)^*$ is a permutation of a word in a^*, b^*, c^* and vice versa. A DTD is called *simple* if all productions in it use simple regular expressions over $E \cup \{\mathtt{S}\}$. Simple regular expressions are prevalent in DTDs. For instance, the Business Process Specification Schema of ebXML [10], a set of specifications to conduct business over the Internet, is a simple DTD. Part of this schema is showed in figure 5.

Theorem 3 *The implication problem for FDs over simple DTDs is solvable in quadratic time.*

In a simple DTD, disjunction can appear in expressions of the form $(a|\epsilon)$ or $(a|b)^*$, but a general disjunction $(a|b)$ is not allowed. We now show that the implication problem remains tractable if the number of such unrestricted disjunctions is small.

A regular expression s over an alphabet A is a *simple disjunction* if $s = \epsilon$, $s = a$, where $a \in A$, or $s = s_1|s_2$, where s_1, s_2 are simple disjunctions over alphabets A_1, A_2 and $A_1 \cap A_2 = \emptyset$. A DTD $D = (E, A, P, R, r)$ is called *disjunctive* if for every $\tau \in E$, $P(\tau) = s_1, \ldots, s_m$, where each s_i is either a simple regular expression or a simple disjunction over an alphabet A_i ($i \in [1, m]$), and $A_i \cap A_j = \emptyset$ ($i, j \in [1, m]$ and $i \neq j$). This generalizes the concept of a simple DTD.

With each disjunctive DTD D, we associate a number N_D that measures the complexity of unrestricted disjunctions in D. Formally, for a simple regular expression s, $N_s = 1$. If s is a simple disjunction, then N_s is the number of symbols $|$ in s plus 1. If $P(\tau) = s_1, \ldots, s_n$, then N_τ is 1, if $s_1, \ldots, s_n$ is a simple regular expression, $N_\tau = |\{p \in \mathit{paths}(D) \mid \mathit{last}(p) = \tau\}| \times N_{s_1} \times \cdots \times N_{s_n}$ otherwise. Finally, $N_D = \prod_{\tau \in E} N_\tau$.

Theorem 4 *For any fixed $k > 0$, the FD implication problem for disjunctive DTDs D with $N_D \leq k \cdot \log(|D|)$ is solvable in polynomial time.* □

There are some classes of DTDs for which the implication problem is not tractable. One such class consists of arbitrary disjunctive DTDs. Another class is that of *relational DTDs*. We say that D is a relational DTD if for each XML tree $T \models D$, if X is a non-empty subset of $\mathit{tuples}_D(T)$, then $\mathit{trees}_D(X) \models D$.

This class contains regular expressions like the one below, from a DTD for Frequently Asked Questions [17]:

```
<!ELEMENT section (logo*, title, (qna+ | q+ |
                        ( p | div | section)+))>
```

There exist non-relational DTDs (for example, `<!ELEMENT a (b,b)>`). However:

Proposition 9 *Every disjunctive DTD is relational.*

Theorem 5 *The FD implication problem over relational DTDs and over disjunctive DTDs is coNP-complete.* □

Relational DTDs have the following useful property that lets us establish the complexity of testing XNF.

Proposition 10 *Given a relational DTD D and a set Σ of FDs over D, (D, Σ) is in XNF iff for each non-trivial FD of the form $S \to p.@l$ or $S \to p.\mathtt{S}$ in Σ, $S \to p \in (D, \Sigma)^+$.* □

From this, we immediately derive:

Corollary 1 *Testing if (D, Σ) is in XNF can be done in cubic time for simple DTDs, and is coNP-complete for relational DTDs.* □

8 Future Research

The decomposition algorithm can be improved in various ways, and we plan to work on making it more efficient. We also would like to find a complete classification of the complexity of the FD implication problem for various classes of DTDs.

As prevalent as BCNF is, it does not solve *all* the problems of relational schema design, and one cannot expect XNF to address all shortcomings of DTD design. We plan to work on extending XNF to more powerful normal forms, in particular by taking into account multivalued dependencies which are naturally induced by the tree structure.

```
<!ELEMENT ProcessSpecification (Documentation*, SubstitutionSet*, (Include | BusinessDocument |
  ProcessSpecification | Package | BinaryCollaboration | BusinessTransaction |
  MultiPartyCollaboration)*)>
<!ELEMENT Include (Documentation*)>
<!ELEMENT BusinessDocument (ConditionExpression?, Documentation*)>
<!ELEMENT SubstitutionSet (DocumentSubstitution | AttributeSubstitution | Documentation)*>
<!ELEMENT BinaryCollaboration (Documentation*, InitiatingRole, RespondingRole, (Documentation |
  Start | Transition | Success | Failure | BusinessTransactionActivity | CollaborationActivity |
  Fork | Join)*)>
<!ELEMENT Transition (ConditionExpression?, Documentation*)>
```

Figure 5: Part of the Business Process Specification Schema of ebXML.

Acknowledgments Discussions with Michael Benedikt and Wenfei Fan were extremely helpful. The authors were supported in part by grants from the Natural Sciences and Engineering Research Council of Canada and from Bell University Laboratories.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] J. Albert, D. Giammarresi, D. Wood. Normal form algorithms for extended context-free grammars. *TCS* 267 (2001), 35–47.

[3] P. Atzeni, N. Morfuni. Functional dependencies in relations with null values. *Information Processing Letters* 18(4): 233–238, (1984).

[4] C. Beeri, P. Bernstein, N. Goodman. A sophisticate's introduction to database normalization theory. *VLDB'78*, pages 113–124.

[5] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *WWW'10*, 2001.

[6] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. In *DBPL'01*.

[7] P. Buneman, A. Jung, A. Ohori, Using powerdomains to generalize relational databases, *Theoretical Computer Science* 91 (1991), 23–55.

[8] DBLP. `http://dblp.uni-trier.de/`.

[9] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *JLP* 1(3): 267–284 (1984).

[10] ebXML. Business Process Specification Schema v1.01. `http://www.ebxml.org/specs/`.

[11] W. Fan, L. Libkin. On XML integrity constraints in the presence of DTDs. In *PODS'01*, pages 114–125.

[12] W. Fan, J. Siméon. Integrity constraints for XML. *PODS'00*, pages 23–34.

[13] M. Fernandez, J. Siméon, P. Wadler. A semi-monad for semi-structured data. *ICDT'01*, pages 263–300.

[14] D. Florescu, D. Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Bull.* 22 (1999), 27–34.

[15] G. Grahne. *The Problem of Incomplete Information in Relational Databases*, Springer, Berlin, 1991.

[16] C. Gunter. *Semantics of Programming Languages*, The MIT Press, 1992.

[17] J. Higgins, R. Jelliffe QAML Version 2.4. `http://xml.ascc.net/resource/qaml-xml.dtd`, 1999.

[18] R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal on Computing* 15(3): 856-886 (1986).

[19] T. Imielinski and W. Lipski. Incomplete information in relational databases. *J. ACM* 31(1984), 761–791.

[20] C. Kanne, G. Moerkotte. Efficient storage of XML data. In *ICDE'00*, p. 198.

[21] M. Levene, G. Loizou. Axiomatisation of functional dependencies in incomplete relations. *TCS* 206(1-2): 283–300, 1998.

[22] W.Y. Mok, Y. K. Ng, D. Embley. A normal form for precisely characterizing redundancy in nested relations. *ACM TODS* 21 (1996), 77–106.

[23] Z. M. Özsoyoglu, L.-Y. Yuan. A new normal form for nested relations. *ACM TODS* 12(1): 111–136, 1987.

[24] Y. Sagiv, C. Delobel, D. S. Parker, R. Fagin. An equivalence between relational database dependencies and a fragment of propositional logic. *J. ACM* 28 (1981), 435–453.

[25] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. DeWitt, J. Naughton. Relational databases for querying XML documents: limitations and opportunities. *VLDB'99*, pages 302–314.

[26] D. Suciu. Bounded fixpoints for complex objects. *TCS* 176 (1997), 283–328.

[27] Z. Tari, J. Stokes, S. Spaccapietra. Object normal forms and dependency constraints for object-oriented schemata. *ACM TODS* 22 (1997), 513–569.

[28] I. Tatarinov, Z. Ives, A. Halevy, D. Weld. Updating XML. In *SIGMOD'01*, pages 413–424.

[29] J. Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *TCS* 254(1-2): 363–377, 2001.

[30] W3C. XML-Data. W3C Note, Jan. 1998.

[31] W3C. XML Schema. W3C Working Draft, May 2001.

[32] W3C. XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001.