A Normal Form for XML Documents

MARCELO ARENAS
University of Toronto
and
LEONID LIBKIN
University of Toronto

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 called NNF-FD when those are appropriately coded as XML documents. Finally, we present a lossless algorithm for converting any DTD into one in XNF.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design—*Data models*; *Normal forms*; *Schema and subschema*; H.2.3 [**Database Management**]: Languages—*Data description languages (DDL)*

General Terms: Design, Management, Theory

Additional Key Words and Phrases: XML data, DTDs, design, normal form, functional dependencies

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 [Kanne and Moerkotte 2000], and update XML data [Tatarinov et al. 2001]. 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

Authors' address: M. Arenas, Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4, e-mail: marenas@cs.toronto.edu; L. Libkin, Department of Computer Science, University of Toronto, 6 King's College Road, Toronto, Ontario, Canada M5S 3H5, e-mail: libkin@cs.toronto.edu.

DTDs (Document Type Definitions), and some other proposals exist or are under development [W3C 2001; 1998]. 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 [Beeri et al. 1978] 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.

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:

```
<!DOCTYPE courses [
  <!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.

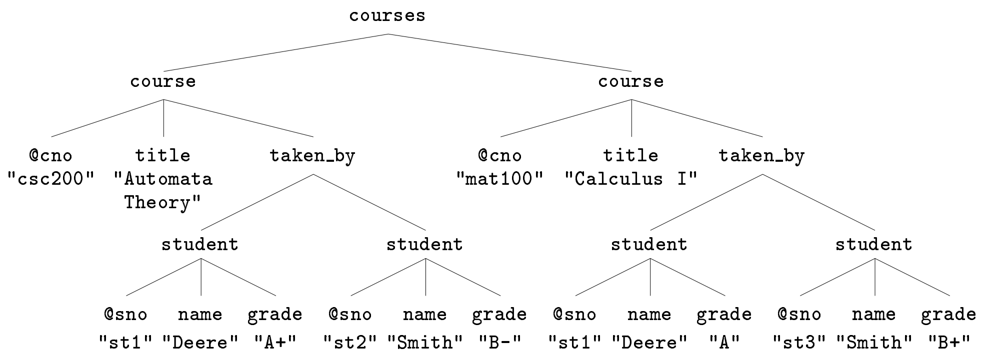


Fig. 1. A document containing redundant information.

An example of an XML document that conforms to this DTD is shown in figure 1. 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:

```
<!DOCTYPE courses [
  <!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 2. Note that `st2` and `st3` are put together because both students have the same name. □

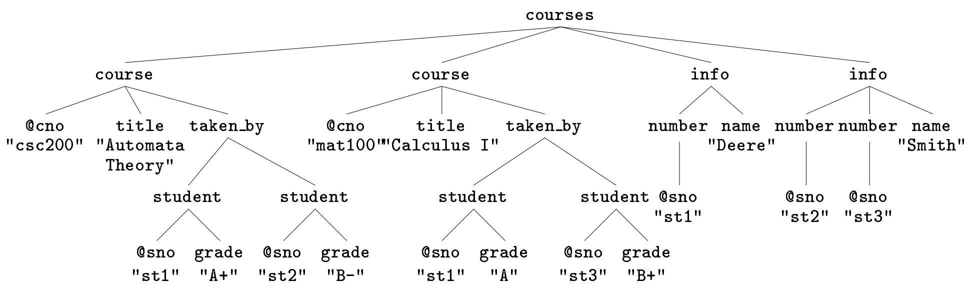


Fig. 2. A well-designed document.

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 [Ley 2003] for storing data about conferences.

```
<!DOCTYPE db [
  <!ELEMENT db (conf*)>
  <!ELEMENT conf (title, issue+)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT issue (inproceedings+)>
  <!ELEMENT inproceedings (author+, title)>
    <!ATTLIST inproceedings
              key ID #REQUIRED
              pages CDATA #REQUIRED
              year CDATA #REQUIRED>
  <!ELEMENT author (#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 [Buneman et al. 2001a; 2001b; W3C 2001]. 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 [Suciu 1997; Van den Bussche 2001]; 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 [Atzeni and Morfuni 1984; Levene and Loizou 1998].

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 [Mok et al. 1996] when only functional dependencies are considered (see Section 5.2 for a precise statement of this claim).

The last step then is to find an algorithm that converts any DTD, given a set of FDs, 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). Finally, in Section 8 we describe related work and some topics of future research.

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 [W3C 2001]. 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.

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 2.1. 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:

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

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 $\mathit{length}(w)$ as n and $\mathit{last}(w)$ as w_n. We let $\mathit{paths}(D)$ stand for the set of all paths in D and $\mathit{EPaths}(D)$ for the set of all paths that ends with an element type (rather than an attribute or S); that is, $\mathit{EPaths}(D) = \{p \in \mathit{paths}(D) \mid \mathit{last}(p) \in E\}$. A DTD is called *recursive* if $\mathit{paths}(D)$ is infinite.

Definition 2.2. An *XML tree* T is defined to be a tree $(V, \mathit{lab}, \mathit{ele}, \mathit{att}, \mathit{root})$, where

—$V \subseteq \mathit{Vert}$ is a finite set of *vertices* (*nodes*).

—$\mathit{lab} : V \to El$.

—$\mathit{ele} : V \to \mathit{Str} \cup V^*$.

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

—$\mathit{root} \in V$ is called *the root of* T.

The parent-child edge relation on V, $\{(v_1, v_2) \mid v_2 \text{ occurs in } \mathit{ele}(v_1)\}$, 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 2.3. 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, $ele(v) = [v_1, \ldots, v_n]$, and 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 $paths(T) \subseteq paths(D)$.

Clearly, $T \models D$ implies T is compatible with 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 [Florescu and Kossmann 1999; Shanmugasundaram et al. 1999], 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, [Buneman et al. 2001a; 2001b; W3C 2001], 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. Since the notion of functional dependency we propose also does not use the ordering in the tree, 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 $T_2 \not\preceq T_1$.

In the following definition we extend the notion of tuple for relational databases to the case of XML. In a relational database, a tuple is a function that assigns to each attribute a value from the corresponding domain. In our setting, a tree tuple t in a DTD D is a function that assigns to each path in D a value in $Vert \cup Str \cup \{\perp\}$ in such a way that t represents a finite tree with paths from D containing at most one occurrence of each path. In this section, we show that an XML tree can be represented as a set of tree tuples.

Definition 3.1 *(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 3.2. Suppose that D is the DTD shown in example 1.1. Then a tree tuple in D assigns values to each path in $paths(D)$:

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

□

We intend to consider tree tuples in XML trees conforming to a DTD. The ability to map a path to null ($\perp$) allow one in principle to consider tuples with paths that do not reach the leaves of a give tree, although our intention is to consider only paths that do reach the leaves. However, nulls are still needed 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 3.3 *($tree_D$)*. 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

—$V = \{v \in Vert \mid \exists p \in paths(D) \text{ such that } v = t.p\}$.

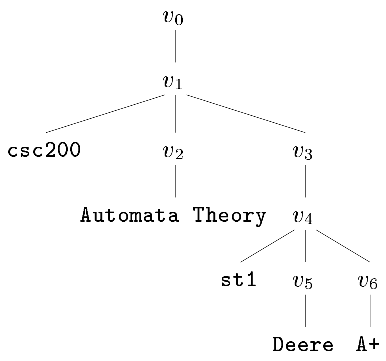
—If $v = t.p$ and $v \in V$, then $lab(v) = last(p)$.

—If $v = t.p$ and $v \in V$, then $ele(v)$ is defined to be the list containing $\{t.p' \mid t.p' \neq \perp \text{ and } p' = p.\tau, \tau \in E, \text{ or } p' = p.\mathtt{S}\}$, ordered lexicographically.

—If $v = t.p$, $@l \in A$ and $t.p.@l \neq \perp$, then $att(v, @l) = t.p.@l$.

We note that in this definition the lexicographic order is arbitrary, and it is chosen simply because an XML tree must be ordered.

Example 3.4. Let D be the DTD from example 1.1 and t the tree tuple from example 3.2. Then, t gives rise to the following XML tree:



□

Notice that the tree in the example conforms to the DTD from example 1.1. In general, this need not be the case. For instance, if the rule <!ELEMENT taken_by (student*)> in the DTD shown in example 1.1 is changed by a rule saying that every course must have at least two students <!ELEMENT taken_by (student, student+)>, then the tree shown in example 3.4 does not conform to the DTD. However, $tree_D(t)$ would always be compatible with D, as easily follows from the definition:

PROPOSITION 3.5. *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, [Buneman et al. 1991; Grahne 1991; Gunter 1992]. 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 3.6 *(tuples$_D$)*. Given a DTD D and an XML tree T such that $T \lhd D$, $\mathit{tuples}_D(T)$ is defined to be the set of maximal, with respect to $\sqsubseteq$, tree tuples t such that $\mathit{tree}_D(t)$ is subsumed by T; that is:

$$\mathit{max}_{\sqsubseteq}\{t \in \mathcal{T}(D) \mid \mathit{tree}_D(t) \preceq T\}.$$

Observe that $T_1 \equiv T_2$ implies $\mathit{tuples}_D(T_1) = \mathit{tuples}_D(T_2)$. Hence, tuples_D applies to equivalence classes: $\mathit{tuples}_D([T]) = \mathit{tuples}_D(T)$. The following proposition lists some simple properties of $\mathit{tuples}_D(\cdot)$.

PROPOSITION 3.7. *If $T \lhd D$, then $\mathit{tuples}_D(T)$ is a finite subset of $\mathcal{T}(D)$. Furthermore, $\mathit{tuples}_D(\cdot)$ is monotone: $T_1 \preceq T_2$ implies $\mathit{tuples}_D(T_1) \sqsubseteq^\flat \mathit{tuples}_D(T_2)$.*

PROOF. We prove only monotonicity. Suppose that $T_1 \preceq T_2$ and $t_1 \in \mathit{tuples}_D(T_1)$. We have to prove that there exists $t_2 \in \mathit{tuples}_D(T_2)$ such that $t_1 \sqsubseteq t_2$. If $t_1 \in \mathit{tuples}_D(T_2)$, this is obvious, so assume that $t_1 \notin \mathit{tuples}_D(T_2)$. Given that $t_1 \in \mathit{tuples}_D(T_1)$, $\mathit{tree}_D(t_1) \preceq T_1$, and, therefore, $\mathit{tree}_D(t_1) \preceq T_2$. Hence, by definition of $\mathit{tuples}_D(\cdot)$, there exists $t_2 \in \mathit{tuples}_D(T_2)$ such that $t_1 \sqsubset t_2$, since $t_1 \notin \mathit{tuples}_D(T_2)$. $\square$

Example 3.8. In example 1.1 we saw a DTD D and a tree T conforming to D. In example 3.2 we saw one tree tuple t for that tree, with identifiers assigned to some of the element nodes of T. If we assign identifiers to the rest of the nodes, we can compute the set $\mathit{tuples}_D(T)$ (the attributes are sorted as in example 3.2):

$$\begin{aligned}\{\ &(v_0,\ v_1,\ \texttt{csc200}, v_2,\ \texttt{Automata Theory},\ v_3,\ v_4,\ \texttt{st1},\ v_5,\ \texttt{Deere},\ v_6,\ \texttt{A+}),\\ &(v_0,\ v_1,\ \texttt{csc200}, v_2,\ \texttt{Automata Theory},\ v_3,\ v_7,\ \texttt{st2},\ v_8,\ \texttt{Smith},\ v_9,\ \texttt{B-}),\\ &(v_0,\ v_{10},\ \texttt{mat100},\ v_{11},\ \texttt{Calculus I},\ v_{12},\ v_{13},\ \texttt{st1},\ v_{14},\ \texttt{Deere},\ v_{15},\ \texttt{A}),\\ &(v_0,\ v_{10},\ \texttt{mat100},\ v_{11},\ \texttt{Calculus I},\ v_{12},\ v_{16},\ \texttt{st3},\ v_{17},\ \texttt{Smith},\ v_{18},\ \texttt{B+})\ \}\end{aligned}$$

$\square$

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 3.9 *(trees$_D$)*. Given a DTD D and a set of tree tuples $X \subseteq \mathcal{T}(D)$, $\mathit{trees}_D(X)$ is defined to be:

$$\mathit{min}_{\preceq}\{T \mid T \lhd D \text{ and } \forall t \in X,\ \mathit{tree}_D(t) \preceq T\}.$$

Notice that if $T \in \mathit{trees}_D(X)$ and $T' \equiv T$, then T' is in $\mathit{trees}_D(X)$. 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 $\mathit{tuples}_D(T)$, up to equivalence $\equiv$.

THEOREM 3.10. *Given a DTD D and an XML tree T, if $T \lhd D$, then $\mathit{trees}_D(\mathit{tuples}_D([T])) = [T]$.*

PROOF. Every XML tree is finite, and, therefore, $tuples_D([T]) = \{t_1, \ldots, t_n\}$, for some n. Suppose that $T \notin trees_D(\{t_1, \ldots, t_n\})$. Given that $tree_D(t_i) \preceq T$, for each $i \in [1, n]$, there is an XML tree T' such that $T' \preceq T$ and $tree_D(t_i) \preceq T'$, for each $i \in [1, n]$. If $T' \prec T$, there is at least one node, string or attribute value contained in T which is not contained in T'. This value must be contained in some tree tuple t_j ($j \in [1, n]$), which contradicts $tree_D(t_j) \preceq T'$. Therefore, $T \in trees_D(tuples_D([T]))$.

Let $T' \in trees_D(tuples_D([T]))$. For each $i \in [1, n]$, $tree_D(t_i) \preceq T'$. Thus, given that $tuples_D(T) = \{t_1, \ldots, t_n\}$, we conclude that $T \preceq T'$, and, therefore, by definition of $trees_D$, $T' \equiv T$. □

Example 3.11. It could be the case that for some set of tree tuples X there is no an XML tree T such that for every $t \in X$, $tree(t) \preceq T$. For example, let D be a DTD $D = (E, A, P, R, r)$, where $E = \{r, a, b\}$, $A = \emptyset$, $P(r) = (a|b)$, $P(a) = \epsilon$ and $P(b) = \epsilon$. Let $t_1, t_2 \in \mathcal{T}(D)$ be defined as

$$\begin{array}{lcllcl} t_1.r & = & v_0 & \quad t_2.r & = & v_2 \\ t_1.r.a & = & v_1 & \quad t_2.r.a & = & \perp \\ t_1.r.b & = & \perp & \quad t_2.r.b & = & v_3 \end{array}$$

Since $t_1.r \neq t_2.r$, there is no an XML tree T such that $tree_D(t_1) \preceq T$ and $tree_D(t_2) \preceq T$. □

We say that $X \subseteq \mathcal{T}(D)$ is *D-compatible* if there is an XML tree T such that $T \lhd D$ and $X \subseteq tuples_D(T)$. For a D-compatible set of tree tuples X there is always an XML tree T such that for every $t \in X$, $tree_D(t) \preceq T$. Moreover,

PROPOSITION 3.12. *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) $X \sqsubseteq^\flat tuples_D(trees_D(X))$.*

PROOF. (a) Assume that $D = (E, A, P, R, r)$. Since X is D-compatible, there exists an XML tree $T' = (V', lab', ele', att', root')$ such that $T' \lhd D$ and $X \subseteq tuples_D(T')$. We use T' to define an XML tree $T = (V, lab, ele, att, root)$ such that $trees_D(X) = [T]$.

For each $v \in V'$, if there is $t \in X$ and $p \in paths(D)$ such that $t.p = v$, then v is included in V. Furthermore, for each $v \in V$, $lab(v)$ is defined as $lab'(v)$, $ele(v) = [s_1, \ldots, s_n]$, where each $s_i = t'.p.\mathtt{S}$ or $s_i = t'.p.\tau$ for some $t' \in X$ and $\tau \in E$ such that $t'.p = v$. For each $@l \in A$ such that $t'.p.@l \neq \perp$ and $t'.p = v$ for some $t' \in X$, $att(v, @l)$ is defined as $t'.p.@l$. Finally, $root$ is defined as $root'$. It is easy to see that $trees_D(X) = [T]$.

(b) Let $t \in X$ and T be an XML tree such that $trees_D(X) = [T]$. If $t \in tuples_D([T])$, then the property holds trivially. Suppose that $t \notin tuples_D([T])$. Then, given that $tree_D(t) \preceq T$, there is $t' \in tuples_D([T])$ such that $t \sqsubset t'$. In either case, we conclude that there is $t' \in tuples_D(trees_D(X))$ such that $t \sqsubseteq t'$. □

The example below shows that it could be the case that $tuples_D(trees_D(X))$ properly dominates X, that is, $X \sqsubseteq^\flat tuples_D(trees_D(X))$ and $tuples_D(trees_D(X)) \not\sqsubseteq^\flat X$. In particular, this example shows that the inverse of Theorem 3.10 does not hold, that is, $tuples_D(trees_D(X))$ is not necessarily equal to X for every set of tree tuples X, even if this set is D-compatible. Let D be as in example 3.11 and $t_1, t_2 \in \mathcal{T}(D)$ be defined as

$$\begin{array}{lcl@{\qquad\qquad}lcl} t_1.r & = & v_0 & t_2.r & = & v_0 \\ t_1.r.a & = & v_1 & t_2.r.a & = & \perp \\ t_1.r.b & = & \perp & t_2.r.b & = & v_2 \end{array}$$

Let t_3 be a tree tuple defined as $t_3.r = v_0$, $t_3.r.a = v_1$ and $t_3.r.b = v_2$. Then, $\mathit{tuples}_D(\mathit{trees}_D(\{t_1, t_2\})) = \{t_3\}$ since $t_1 \sqsubset t_3$ and $t_2 \sqsubset t_3$, and, therefore, $\{t_1, t_2\} \sqsubseteq^\flat \mathit{tuples}_D(\mathit{trees}_D(\{t_1, t_2\}))$ and $\mathit{tuples}_D(\mathit{trees}_D(\{t_1, t_2\})) \not\sqsubseteq^\flat \{t_1, t_2\}$.

From Theorem 3.10 and Proposition 3.12, it is straightforward to prove the following Corollary.

Corollary 3.13. *For a D-compatible set of tree tuples X, $\mathit{trees}_D(\mathit{tuples}_D(\mathit{trees}_D(X))) = \mathit{trees}_D(X)$.*

Theorem 3.10 and Proposition 3.12 are summarized in the diagram presented in the following figure. In this diagram, X is a D-compatible set of tree tuples. The arrow $\hookrightarrow$ stands for the $\sqsubseteq^\flat$ ordering.

$$\begin{tikzcd} X \arrow[rr, "\mathit{trees}_D"] \arrow[drr, hook] & & {[T]} \arrow[d, shift right, "\mathit{tuples}_D"'] \\ & & X' \arrow[u, shift right, "\mathit{trees}_D"'] \end{tikzcd}$$

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 $S_1 \to S_2$ where S_1, S_2 are finite non-empty subsets of $\mathit{paths}(D)$. The set of all FDs over D is denoted by $\mathcal{FD}(D)$.

For $S \subseteq \mathit{paths}(D)$, 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 $S_1 \to S_2 \in \mathcal{FD}(D)$ and T is an XML tree such that $T \lhd D$ and $S_1 \cup S_2 \subseteq \mathit{paths}(T)$, we say that T *satisfies* $S_1 \to S_2$ (written $T \models S_1 \to S_2$) if for every $t_1, t_2 \in \mathit{tuples}_D(T)$, $t_1.S_1 = t_2.S_1$ and $t_1.S_1 \neq \perp$ imply $t_1.S_2 = t_2.S_2$. We observe that if tree tuples t_1, t_2 satisfy an FD $S_1 \to S_2$, then for every path $p \in S_2$, $t_1.p$ and $t_2.p$ are either both null or both non-null. Moreover, if for every pair of tree tuples t_1, t_2 in an XML tree T, $t_1.S_1 = t_2.S_1$ implies they have a null value on some $p \in S_1$, then the FD is trivially satisfied by T.

The previous definition 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`:

$$\mathit{courses.course.@cno} \to \mathit{courses.course}. \tag{FD1}$$

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

$$courses.course.taken\_by.student.@sno \rightarrow courses.course.taken\_by.student.name.\mathtt{S}. \quad \text{(FD3)}$$

□

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 [Atzeni and Morfuni 1984; Levene and Loizou 1998].

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 $(D, \Sigma)^+$. Furthermore, an FD φ is *trivial* if $(D, \emptyset) \vdash \varphi$. In relational databases, the only trivial FDs are $X \rightarrow Y$, with $Y \subseteq X$. Here, DTD forces some more interesting trivial FDs. For instance, for each $p \in \mathit{EPaths}(D)$ and p' a prefix of p, $(D, \emptyset) \vdash p \rightarrow p'$, and for every p, $p.@l \in \mathit{paths}(D)$, $(D, \emptyset) \vdash p \rightarrow p.@l$. As a matter of fact, trivial functional dependencies in XML documents can be much more complicated than in the relational case, as we show in the following example.

Example 4.2. Let $D = (E, A, P, R, r)$ be a DTD. Assume that a, b and c are element types in D and $P(r) = (a|b|c)$. Then, for every $p \in \mathit{paths}(D)$, $\{r.a, r.b\} \rightarrow p$ is a trivial FD since for every XML tree T conforming to D and every tree tuple t in T, $t.r.a = \perp$ or $t.r.b = \perp$. □

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

The intuition is as follows. Suppose that $S \rightarrow p.@l$ 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, $S \rightarrow p$ must be implied by (D, Σ).

In this definition, we impose the condition that φ is a nontrivial 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 [Mok et al. 1996; Özsoyoglu and Yuan 1987], when only functional dependencies are provided.

Example 5.2. Consider the DTD from example 1.1 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 functional dependency

$$courses.course.taken\_by.student.@sno \ \to \ courses.course.taken\_by.student.name.$$

To remedy this, we gave a revised DTD in example 1.1. The idea was to create a new element `info` for storing information about students. 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 5.3. Suppose that D is the DBLP DTD from example 1.2. Among the set Σ of FDs satisfied by the documents are:

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

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

$$\{db.conf.issue,\ db.conf.issue.inproceedings.title.\mathtt{S}\} \to db.conf.issue.inproceedings \qquad \text{(FD6)}$$

$$db.conf.issue.inproceedings.@key \to db.conf.issue.inproceedings \qquad \text{(FD7)}$$

Constraint (FD4) enforces that two distinct conferences have distinct titles. Given that an issue of a conference represents a particular year of the conference, constraint (FD5) enforces that two articles of the same issue must have the same value in the attribute year. Constraint (FD6) enforces that for a given issue of a conference, two distinct articles must have different titles. Finally, constraint (FD7) enforces that `key` is an identifier for each article in the database.

By (FD5) for each issue of a conference, its year is stored in every article in that issue and, thus, DBLP documents can store redundant information. (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. □

5.1 BCNF and XNF

Relational databases can be easily mapped into XML documents. Given a relation $G(A_1, \ldots, A_n)$ and a set of FDs *FD* over G, we translate the schema (G, FD) into an XML representation, that is, a DTD and a set of XML FDs (D_G, Σ_{FD}). The DTD $D_G = (E,\, A,\, P,\, R,\, db)$ is defined as follows:

—$E = \{db, G\}$.
—$A = \{@A_1, \ldots, @A_n\}$.
—$P(db) = G^*$ and $P(G) = \epsilon$.
—$R(db) = \emptyset$, $R(G) = \{@A_1, \ldots, @A_n\}$.

Without loss of generality, assume that all FDs are of the form $X \to A$, where A is an attribute. Then Σ_{FD} over D_G is defined as follows.

—For each FD $A_{i_1} \cdots A_{i_m} \to A_i \in FD$, $\{db.G.@A_{i_1},\, \ldots,\, db.G.@A_{i_m}\} \to db.G.@A_i$ is in Σ_{FD}.
—$\{db.G.@A_1,\, \ldots,\, db.G.@A_n\} \to db.G$ is in Σ_{FD}.

The latter is included to avoid duplicates.

Example 5.4. 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 $db.G.@A \to db.G.@B$. □

The following proposition shows that BCNF and XNF are equivalent when relational databases are appropriately coded as XML documents.

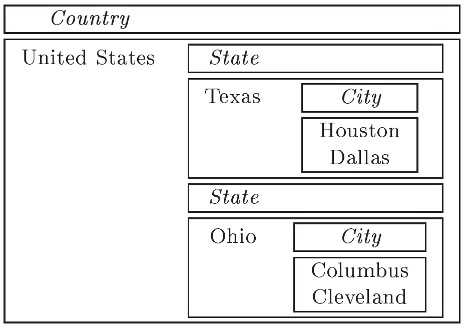
PROPOSITION 5.5. *Given a relation schema $G(A_1, \ldots, A_n)$ and a set of functional dependencies FD over G, (G, FD) is in BCNF iff (D_G, Σ_{FD}) is in XNF.*

PROOF. This follows from Proposition 5.6 (to be proved in the next section) since every relation schema is trivially consistent (see next section) and NNF-FD coincides with BCNF when only functional dependencies are provided [Mok et al. 1996]. □

5.2 NNF and XNF

A nested relation schema is either a set of attributes X, or $X(G_1)^* \ldots (G_n)^*$, where G_i's are nested relation 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 nested schema $G = X(G_1)^* \ldots (G_n)^*$, we introduce an element type G with $P(G) = G_1^*, \ldots, G_n^*$ and $R(G) = \{@A_1, \ldots, @A_m\}$, where $X = \{A_1, \ldots, A_m\}$; 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:



(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

Fig. 3. Nested relation and its unnesting.

```
<!DOCTYPE db [
  <!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) [Abiteboul et al. 1995; Mok et al. 1996; Özsoyoglu and Yuan 1987]. A nested relation r over $X(G_1)^* \ldots (G_n)^*$ 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 $t_1.G_i$ is 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 $G_2, \ldots, G_n$ 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 $A_{i_1} \cdots A_{i_m} \to A_i \in FD$, $\{path(A_{i_1}),\ \ldots,\ path(A_{i_m})\} \to path(A_i)$ is in Σ_{FD}.

—For each $i \in [1, n]$, if $A_{j_1}, \ldots, A_{j_m}$ is the set of atomic attributes of G_i and G_i is a nested attribute of G_j, $\{path(G_j),\ path(A_{j_1}),\ \ldots,\ path(A_{j_m})\} \to path(G_i)$ is in Σ_{FD}.

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

Note that the last rule imposes the partition normal form. The set Σ_{PNF} contains all the functional dependencies defined by this rule.

Normal forms for nested relations were proposed in [Mok et al. 1996; Özsoyoglu and Yuan 1987]. These normal forms were defined for nested schemas containing functional and multivalued dependencies. Here we consider a normal form NNF-FD, which is the nested normal form NNF introduced in [Mok et al. 1996] restricted to FDs only. To define this normal form we need to introduce some terminology.

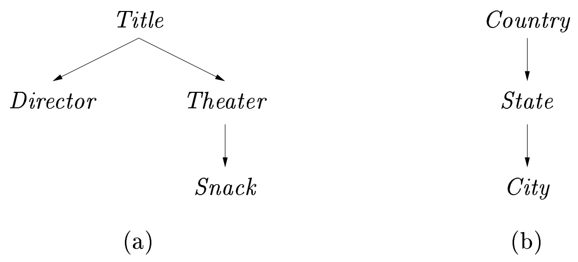


Fig. 4. Two schema trees.

Every nested relation schema G can be represented as a tree $st(G)$, called the *schema tree of* G. Formally, if G is a flat schema containing a set of attributes X, then $st(G)$ is a single node tree whose root is the set of attributes X. Otherwise, G is of the form $X(G_1)^* \ldots (G_n)^*$ and $st(G)$ is a tree defined as follows. The root of $st(G)$ is X and the children of X are the roots of $st(G_1), \ldots, st(G_n)$. For example, the schema trees of nested relation schemas $G_1 = Title(G_2)^*(G_3)^*$, $G_2 = Director$, $G_3 = Theater(G_4)^*$, $G_4 = Snack$ and $H_1 = Country(H_2)^*$, $H_2 = State(H_3)^*$, $H_3 = City$ are shown in figures 4 (a) and 4 (b), respectively. Given a nested relation schema G including a set of attributes U, for each node X of $st(G)$ we define $ancestor(X)$ as the union of attributes in all ancestors of X in $st(G)$, including X. For instance, $ancestor(State) = \{Country, State\}$ in the schema tree shown in figure 4 (b). Similarly, for every $A \in U$, we define $ancestor(A)$ as the set of attributes $ancestor(X_A)$, where X_A is the one of $st(G)$ containing the attribute A, and for every node X of $st(G)$ we define $descendant(X)$ as the union of attributes in all descendants of X in $st(G)$, including X.

Data dependencies for nested relations are defined by using the notion of complete unnesting. Thus, every nested schema has some multivalued dependencies. For example, the nested relation schema $G_1 = \mathit{Title}(G_2)^*(G_3)^*$, $G_2 = \mathit{Director}$, $G_3 = \mathit{Theater}(G_4)^*$, $G_4 = \mathit{Snack}$ has the following set of multivalued dependencies:

$$\{\mathit{Title} \twoheadrightarrow \mathit{Director},\ \mathit{Title} \twoheadrightarrow \{\mathit{Theater}, \mathit{Snack}\},\ \{\mathit{Title}, \mathit{Theater}\} \twoheadrightarrow \mathit{Snack}\},$$

since for every nested relation I of G_1, the complete unnesting of I satisfies these dependencies. Formally, the set of multivalued dependencies embedded in a nested relation schema G is defined to be:

$$\mathit{MVD}(G) = \{\mathit{ancestor}(X) \twoheadrightarrow \mathit{descendant}(Y) \mid (X, Y) \text{ is an edge in } \mathit{st}(G)\}.$$

Given a nested relation schema G, the set $\mathit{MVD}(G)$ is used to define NNF-FD. More precisely, if FD is a set of FDs over G, then (G, FD) is in NNF-FD [Mok et al. 1996] if (1) $\mathit{FD} \vdash \mathit{MVD}(G)$, that is, every multivalued dependency embedded in G is implied by FD, and (2) for each nontrivial FD $X \to A \in (G, \mathit{FD})^+$, $X \to \mathit{ancestor}(A)$ is also in $(G, \mathit{FD})^+$. As before, $(G, \mathit{FD})^+$ stands for the set of all FDs implied by (G, FD).

To establish the relationship between NNF-FD and XNF, we have to introduce the notion of consistent nested schemas. Given a nested relation schema G and a set of FDs FD over G, (G, FD) is *consistent* [Mok et al. 1996] if $\mathit{FD} \vdash \mathit{MVD}(G)$. It was shown in [Mok et al. 1996] that for consistent nested schemas, NNF precisely characterize redundancy in nested relations. The result below shows that for consistent nested schemas, NNF-FD and XNF coincide.

Proposition 5.6. *Let G be a nested relation schema and FD a set of functional dependencies over G such that (G, FD) is consistent. Then (G, FD) is in NNF-FD iff (D_G, Σ_{FD}) is in XNF.*

Proof. First we need to prove the following claim.

Claim 5.7. *$A_{i_1} \cdots A_{i_m} \to A_i \in (G, \mathit{FD})^+$ if and only if $\{\mathit{path}(A_{i_1}), \ldots, \mathit{path}(A_{i_m})\} \to \mathit{path}(A_i) \in (D_G, \Sigma_{FD})^+$.*

The proof of this claim follows from the following fact. For each instance I of G, there is an XML tree T_I conforming to D_G such that $I \models \mathit{FD}$ iff $T_I \models \Sigma_{FD}$. Moreover, for each XML tree T conforming to D_G and satisfying Σ_{PNF}, there is an instance I_T of G such that $T \models \Sigma_{FD}$ iff $I_T \models \mathit{FD}$.

Now we prove the proposition.

($\Leftarrow$) Suppose that (D_G, Σ_{FD}) is in XNF. We prove that (G, FD) is in NNF-FD. Given that (G, FD) is consistent, we only need to consider the second condition in the definition of NNF-FD. Let $A_{i_1} \cdots A_{i_m} \to A_i$ be a nontrivial functional dependency in $(G, \mathit{FD})^+$. We have to prove that $A_{i_1}, \ldots, A_{i_m} \to \mathit{ancestor}(A_i)$ is in $(G, \mathit{FD})^+$. By Claim 5.7, we know that $\{\mathit{path}(A_{i_1}), \ldots, \mathit{path}(A_{i_m})\} \to \mathit{path}(A_i)$ is a nontrivial functional dependency in $(D_G, \Sigma_{FD})^+$. Since (D_G, Σ_{FD}) is in XNF, $\{\mathit{path}(A_{i_1}), \ldots, \mathit{path}(A_{i_m})\} \to \mathit{path}(G_j)$ is in $(D_G, \Sigma_{FD})^+$, where G_j is a nested relation schema contained in G such that A_i is an atomic attribute of G_j. Thus, given that $\mathit{path}(G_j) \to \mathit{path}(A)$ is a trivial functional dependency in D_G, for each $A \in \mathit{ancestor}(A_i)$, we conclude that $\{\mathit{path}(A_{i_1}), \ldots, \mathit{path}(A_{i_m})\} \to$

$path(A)$ is in $(D_G, \Sigma_{FD})^+$ for each $A \in ancestor(A_i)$. By Claim 5.7, $A_{i_1} \cdots A_{i_m} \to ancestor(A_i)$ is in $(G, FD)^+$.

($\Rightarrow$) Suppose that (G, FD) is in NNF-FD. We will prove that (D_G, Σ_{FD}) is in XNF. Let R be a nested relation schema contained in G and A an atomic attribute of R. Suppose that there is $S \subseteq paths(D_G)$ such that $S \to path(A)$ is a nontrivial functional dependency in $(D_G, \Sigma_{FD})^+$. We have to prove that $S \to path(R) \in (D_G, \Sigma_{FD})^+$. Let S_1 and S_2 be set of paths such that $S = S_1 \cup S_2$, $S_1 \subseteq EPaths(D_G)$ and $S_2 \cap EPaths(D_G) = \emptyset$. Let $S_1' = \{path(A') \mid$ there is $path(R') \in S_1$ such that A' is an atomic attribute of some nested relation schema mentioned in $path(R')\}$. Given that $\Sigma_{PNF} \subseteq \Sigma_{FD}$, $S_1' \to S_1 \in (D_G, \Sigma_{FD})^+$. Thus, $S_1' \cup S_2 \to path(A) \in (D_G, \Sigma_{FD})^+$. Assume that $S_1' \cup S_2 = \{path(A_{i_1}), \ldots, path(A_{i_m})\}$. By Claim 5.7, $A_{i_1} \cdots A_{i_m} \to A$ is a nontrivial functional dependency in $(G, FD)^+$. Thus, given that (G, FD) is in NNF-FD, we conclude that $A_{i_1} \cdots A_{i_m} \to ancestor(A)$ is in $(G, FD)^+$. Therefore, by Claim 5.7, $S_1' \cup S_2 \to path(B)$ is in $(D_G, \Sigma_{FD})^+$, for each $B \in ancestor(A)$. But $\{path(B) \mid B \in ancestor(A)\} \to path(R)$ is in $(D_G, \Sigma_{FD})^+$, since $\Sigma_{PNF} \subseteq \Sigma_{FD}$. Thus, $S_1' \cup S_2 \to path(R) \in (D_G, \Sigma_{FD})^+$, and given that $S_1 \to S_1'$ is a trivial functional dependency in D_G, we conclude that $S \to path(R)$ is in $(D_G, \Sigma_{FD})^+$. $\square$

6. NORMALIZATION ALGORITHMS

The goal of this section is to 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. This can be done without any loss of generality. Notice that in a recursive DTD D, the set of all paths is infinite. However, a given set of FDs Σ only mentions a finite number of paths, which means that it suffices to restrict one's attention to a finite number of "unfoldings" of recursive rules.

We make an additional assumption that all the FDs are of the form: $\{q, p_1.@l_1, \ldots, p_n.@l_n\} \to p$. 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 (in fact it is very hard to come up with a reasonable example where they could be used). Furthermore, even if we have such constraints, they can be easily eliminated. To do so, we create a new attribute $@l$, remove $\{q, q'\} \cup S \to p$ and replace it by $q'.@l \to q'$ and $\{q, q'.@l\} \cup S \to p$.

We shall also assume that paths do not contain the symbol S (since p.S can always be replaced by a path of the form $p.@l$).

6.1 The Decomposition Algorithm

For presenting the algorithm and proving its losslessness, we make the following assumption: if $X \to p.@l$ is an FD that causes a violation of XNF, then every time that $p.@l$ is not null, every path in X is not null. This will make our presentation simpler, and then at the end of the section we will show how to eliminate this assumption.

Given a DTD D and a set of FDs Σ, a nontrivial FD $S \to p.@l$ is called *anomalous*, over (D, Σ), if it violates XNF; that is, $S \to p.@l \in (D, \Sigma)^+$ but

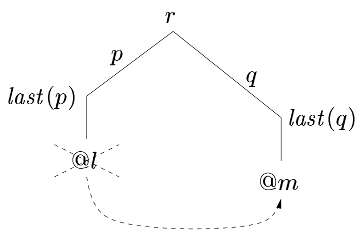
$S \rightarrow p \notin (D, \Sigma)^+$. A path on the right-hand side of an anomalous FD is called an anomalous path, and the set of all such paths is denoted by $AP(D, \Sigma)$.

In this section we present an XNF decomposition algorithm that combines two basic ideas presented in the introduction: creating a new element type, and moving an attribute.

6.1.1 *Moving attributes.* Let $D = (E, A, P, R, r)$ be a DTD and Σ a set of FDs over D. Assume that (D, Σ) contains an anomalous FD $q \rightarrow p.@l$, where $q \in EPaths(D)$. For example, the DBLP database shown in example 1.2 contains an anomalous FD of this form:

$$db.conf.issue \rightarrow db.conf.issue.inproceedings.@year. \tag{1}$$

To eliminate the anomalous FD, we move the attribute $@l$ from the set of attributes of the last element of p to the set of attributes of the last element of q, as shown in the following figure.



For instance, to eliminate the anomalous functional dependency (1) we move the attribute *@year* from the set of attributes of *inproceedings* to the set of attributes of *issue*. Formally, the new DTD $D[p.@l := q.@m]$, where $@m$ is an attribute, is defined to be (E, A', P, R', r), where $A' = A \cup \{@m\}$, $R'(last(q)) = R(last(q)) \cup \{@m\}$, $R'(last(p)) = R(last(p)) - \{@l\}$ and $R'(\tau') = R(\tau')$ for each $\tau' \in E - \{last(q), last(p)\}$.

After transforming D into a new DTD $D[p.@l := q.@m]$, a new set of functional dependencies is generated. Formally, the 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 $S_1 \cup S_2 \subseteq paths(D[p.@l := q.@m])$. Observe that the new set of FDs does not include the functional dependency $q \rightarrow p.@l$ and, thus, it contains a smaller number of anomalous paths, as we show in the following proposition.

PROPOSITION 6.1. *Let D be a DTD, Σ a set of FDs over D, $q \rightarrow p.@l$ an anomalous FD, with $q \in EPaths(D)$, $D' = D[p.@l := q.@m]$, where $@m$ is not an attribute of $last(q)$, and $\Sigma' = \Sigma[p.@l := q.@m]$. Then $AP(D', \Sigma') \subsetneq AP(D, \Sigma)$.*

PROOF. First, we prove (by contradiction) that $q.@m \notin AP(D', \Sigma')$. Suppose that $S' \subseteq paths(D')$ and $S' \rightarrow q.@m \in (D', \Sigma')^+$ is a nontrivial functional dependency. Assume that $S' \rightarrow q \notin (D', \Sigma')^+$. Then there is an XML tree T' such that $T' \models (D', \Sigma')$ and T' contains tree tuples t_1, t_2 such that $t_1.S' = t_2.S'$, $t_1.S' \neq \perp$ and $t_1.q \neq t_2.q$. Given that there is no a constraint in Σ' including the path $q.@m$, the XML tree T'' constructed from T' by giving two distinct values to $t_1.q.@m$ and $t_2.q.@m$ conforms to D', satisfies Σ' and does not satisfy $S' \rightarrow q.@m$, a contradiction. Hence, $q.@m \notin AP(D', \Sigma')$.

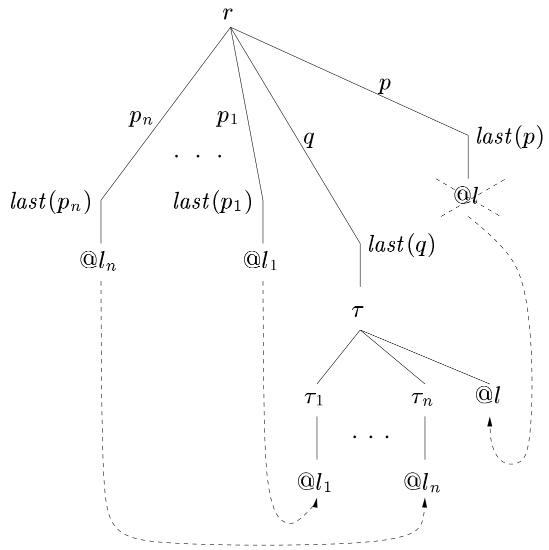
Second, we prove that for every $S_1 \cup S_2 \subseteq \mathit{paths}(D') - \{q.@m\}$, $(D, \Sigma) \vdash S_1 \to S_2$ if and only if $(D', \Sigma') \vdash S_1 \to S_2$, and, thus, by considering the previous paragraph we conclude that $AP(D', \Sigma') \subseteq AP(D, \Sigma)$. Let $S_1 \cup S_2 \subseteq \mathit{paths}(D') - \{q.@m\}$. By definition of Σ', we know that if $(D, \Sigma) \vdash S_1 \to S_2$, then $(D', \Sigma') \vdash S_1 \to S_2$ and, therefore, we only need to prove the other direction. Assume that $(D, \Sigma) \nvdash S_1 \to S_2$. Then there exists an XML tree T such that $T \models (D, \Sigma)$ and $T \not\models S_1 \to S_2$. Define an XML tree T' from T by assigning arbitrary values to $q.@m$ and removing the attribute $@l$ from $\mathit{last}(p)$. Then $T' \models (D', \Sigma')$ and $T' \not\models S_1 \to S_2$, since all the paths mentioned in $\Sigma' \cup \{S_1 \to S_2\}$ are included in $\mathit{paths}(D') - \{q.@m\}$. Thus, $(D', \Sigma') \nvdash S_1 \to S_2$.

To conclude the proof we note that $p.@l \in AP(D, \Sigma)$ and $p.@l \notin AP(D', \Sigma')$, since $p.@l \notin \mathit{paths}(D')$. Therefore, $AP(D', \Sigma') \subsetneqq AP(D, \Sigma)$. □

6.1.2 *Creating new element types.* Let $D = (E,\ A,\ P,\ R,\ r)$ be a DTD and Σ a set of FDs over D. Assume that (D, Σ) contains an anomalous FD $\{q, p_1.@l_1, \ldots, p_n.@l_n\} \to p.@l$, where $q \in \mathit{EPaths}(D)$ and $n \geq 1$. For example, the university database shown in example 1.1 contains an anomalous FD of this form (considering *name*.S as an attribute of *student*):

$$\{\mathit{courses},\ \mathit{courses.course.taken\_by.student.@sno}\} \to \mathit{courses.course.taken\_by.student.name}.\mathsf{S}. \quad (2)$$

To eliminate the anomalous FD, we create a new element type τ as a child of the last element of q, we make $\tau_1, \ldots, \tau_n$ its children, where $\tau_1, \ldots, \tau_n$ are new element types, we remove $@l$ from the list of attributes of $\mathit{last}(p)$ and we make it an attribute of τ and we make $@l_1, \ldots, @l_n$ attributes of $\tau_1, \ldots, \tau_n$, respectively, but without removing them from the sets of attributes of $\mathit{last}(p_1), \ldots, \mathit{last}(p_n)$, as shown in the following figure.



For instance, to eliminate the anomalous functional dependency (2), in example 1.1 we create a new element type *info* as a child of *courses*, we remove *name*.S from *student* and we make it an "attribute" of *info*, we create an element type *number* as a child of *info* and we make *@sno* its attribute. We note that we do not remove *@sno* as an attribute of *student*. Formally, if τ, τ_1, ..., τ_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) if $P(last(q))$ is a regular expression s, then $P'(last(q))$ is defined as the concatenation of s and τ^*, that is (s, τ^*). Furthermore, $P'(\tau)$ is defined as the concatenation of $\tau_1^*, \ldots, \tau_n^*$, $P'(\tau_i) = \epsilon$, for each $i \in [1, n]$, and $P'(\tau') = P(\tau')$, for each $\tau' \in E - \{last(q)\}$.
(2) $R'(\tau) = \{@l\}$, $R'(\tau_i) = \{@l_i\}$, for each $i \in [1, n]$, $R'(last(p)) = R(last(p)) - \{@l\}$ and $R'(\tau') = R(\tau')$ for each $\tau' \in E - \{last(p)\}$.

After transforming D into a new DTD $D' = D[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$, a new set of functional dependencies is generated. Formally, $\Sigma[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$ is a set of FDs over D' defined as the union of the sets of constraints defined in 1., 2. and 3.:

(1) $S_1 \to S_2 \in (D, \Sigma)^+$ 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 $S_1 \to S_2 \in (D, \Sigma)^+$, then we include an FD obtained from $S_1 \to S_2$ by changing p_i to $q.\tau.\tau_i$, $p_i.@l_i$ to $q.\tau.\tau_i.@l_i$, and $p.@l$ to $q.\tau.@l$.
(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].

We are not interested in applying this transformation to an arbitrary anomalous FD, but rather to a *minimal* one. To understand the notion of minimality for XML FDs, we first introduce this notion for relational databases. Let R be a relation schema containing a set of attributes U and Σ be a set of FDs over R. If (R, Σ) is not in BCNF, then there exist pairwise disjoint sets of attributes X, Y and Z such that $U = X \cup Y \cup Z$, $\Sigma \vdash X \to Y$ and $\Sigma \nvdash X \to A$, for every $A \in Z$. In this case we say that $X \to Y$ is an anomalous FD. To eliminate this anomaly, a decomposition algorithm splits relation R into two relations: $S(X, Y)$ and $T(X, Z)$. A desirable property of the new schema is that S or T is in BCNF. We say that $X \to Y$ is a minimal anomalous FD if $S(X, Y)$ is in BCNF, that is, $S(X, Y)$ does not contain an anomalous FD. This condition can be defined as follows: $X \to Y$ is *minimal* if there are no pairwise disjoint sets $X', Y' \subseteq U$ such that $X' \cup Y' \subsetneqq X \cup Y$, $\Sigma \vdash X' \to Y'$ and $\Sigma \nvdash X' \to X \cup Y$.

In the XML context, the definition of minimality is similar in the sense that we expect the new element types τ, τ_1, ..., τ_n form a structure not containing anomalous elements. However, the definition of minimality is 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

[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.

$i \in [0, n]$ 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 6.2. *Let D be a DTD, Σ a set of FDs over D and $\{q, p_1.@l_1, \ldots, p_n.@l_n\} \rightarrow p.@l$ a (D, Σ)-minimal anomalous FD, where $q \in EPaths(D)$ and $n \geq 1$. If $D' = D[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$, where τ, τ_1, ..., τ_n are new element types, and $\Sigma' = \Sigma[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$, then $AP(D', \Sigma') \subsetneq AP(D, \Sigma)$.*

PROOF. First, we prove that $q.\tau.\tau_i.@l_i \notin AP(D', \Sigma')$, for each $i \in [1, n]$. Suppose that there is $S' \subseteq paths(D')$ such that $S' \rightarrow q.\tau.\tau_i.@l_i$ is a nontrivial functional dependency in $(D', \Sigma')^+$ for some $i \in [1, n]$. Notice that $q.\tau.\tau_i \notin S'$, since $q.\tau.\tau_i \rightarrow q.\tau.\tau_i.@l_i$ is a trivial functional dependency. Let $S_1 \cup S_2 = S'$, where (1) $S_1 \cap (\{q, q.\tau.@l\} \cup \{q.\tau.\tau_j \mid j \in [1, n] \text{ and } j \neq i\} \cup \{q.\tau.\tau_j.@l_j \mid j \in [1, n]\}) = \emptyset$ and (2) $S_2 \subseteq \{q, q.\tau.@l\} \cup \{q.\tau.\tau_j \mid j \in [1, n] \text{ and } j \neq i\} \cup \{q.\tau.\tau_j.@l_j \mid j \in [1, n]\}$.

If there is no an XML tree T' conforming to D', satisfying Σ' and containing a tuple t such that $t.S_1 \cup S_2 \neq \perp$, then $S_1 \cup S_2 \rightarrow q.\tau.\tau_i$ must be in $(D', \Sigma')^+$. In this case $q.\tau.\tau_i.@l_i \notin AP(D', \Sigma')$. Suppose that there is an XML tree T' conforming to D', satisfying Σ' and containing a tuple t such that $t.S_1 \cup S_2 \neq \perp$. In this case, by definition of Σ' it is straightforward to prove that $S_2 \rightarrow q.\tau.\tau_i.@l_i$ is in $(D', \Sigma')^+$.

By definition of Σ' and (D, Σ)-minimality of $\{q, p_1.@l_1, \ldots, p_n.@l_n\} \rightarrow p.@l$, one of the following is true: (1) $S_2 \rightarrow q.\tau.\tau_i.@l_i$ is not an anomalous FD, (2) $\{q, q.\tau.\tau_1.@l_1, \ldots, q.\tau.\tau_n.@l_n, q.\tau.@l\} = S_2 \cup \{q.\tau.\tau_i.@l_i\}$ or (3) $\{q.\tau.\tau_j, q.\tau.\tau_1.@l_1, \ldots, q.\tau.\tau_n.@l_n, q.\tau.@l\} = S_2 \cup \{q.\tau.\tau_i.@l_i\}$ for some $j \neq i$ ($j \in [1, n]$). In the first case, $q.\tau.\tau_i.@l_i \notin AP(D', \Sigma')$, so we assume that either (2) or (3) holds. We prove that $S_2 \rightarrow q.\tau.\tau_i$ must be in $(D', \Sigma')^+$. If either (2) or (3) holds, then $S_2 \cup \{q.\tau.\tau_i.@l_i\} \rightarrow q.\tau$ is in $(D', \Sigma')^+$ since $\{q, q.\tau.\tau_1.@l_1, \ldots, q.\tau.\tau_n.@l_n\} \rightarrow q.\tau$ is in Σ' and $q.\tau.\tau_k \rightarrow q$ is a trivial FD in D', for every $k \in [1, n]$. Let T' be an XML tree conforming to D' and satisfying Σ' and $t_1, t_2 \in tuples_{D'}(T')$ such that $t_1.S_2 = t_2.S_2$ and $t_1.S_2 \neq \perp$. Given that $S_2 \rightarrow q.\tau.\tau_i.@l_i \in (D', \Sigma')^+$, $t_1.q.\tau.\tau_i.@l_i = t_2.q.\tau.\tau_i.@l_i$. If $t_1.q.\tau.\tau_i.@l_i = \perp$, then $t_1.q.\tau.\tau_i = t_2.q.\tau.\tau_i = \perp$. If $t_1.q.\tau.\tau_i.@l_i \neq \perp$, then $t_1.q.\tau = t_2.q.\tau$ and $t_1.q.\tau \neq \perp$, because $S_2 \cup \{q.\tau.\tau_i.@l_i\} \rightarrow q.\tau \in (D', \Sigma')^+$. But, by definition of Σ', $\{q.\tau, q.\tau.\tau_i.@l_i\} \rightarrow q.\tau.\tau_i \in \Sigma'$, and, therefore, $t_1.q.\tau.\tau_i = t_2.q.\tau.\tau_i$. In any case, we conclude that $t_1.q.\tau.\tau_i = t_2.q.\tau.\tau_i$ and, therefore, $S_2 \rightarrow q.\tau.\tau_i \in (D', \Sigma')^+$. Thus, $q.\tau.\tau_i.@l_i \notin AP(D', \Sigma')$.

In a similar way, we conclude that $q.\tau.@l \notin AP(D', \Sigma')$.

Second, we prove that for every $S_3 \cup S_4 \subseteq paths(D) - \{p.@l\}$, $(D, \Sigma) \vdash S_3 \rightarrow S_4$ if and only if $(D', \Sigma') \vdash S_3 \rightarrow S_4$, and, thus, by considering the previous paragraph we conclude that $AP(D', \Sigma') \subseteq AP(D, \Sigma)$. Let $S_3 \cup S_4 \subseteq paths(D) - \{p.@l\}$. By definition of Σ', we know that if $(D, \Sigma) \vdash S_3 \rightarrow S_4$, then $(D', \Sigma') \vdash S_3 \rightarrow S_4$ and, therefore, we only need to prove the other direction. Assume that $(D, \Sigma) \nvdash S_3 \rightarrow S_4$. Then there exists an XML tree T such that $T \models (D, \Sigma)$ and $T \not\models S_3 \rightarrow S_4$. Define an XML tree T' from T by assigning $\perp$ to $q.\tau$ and removing the attribute $@l$ from $last(p)$. Then $T' \models (D', \Sigma')$ and $T' \not\models S_3 \rightarrow S_4$, since all the paths mentioned in $\Sigma' \cup \{S_3 \rightarrow S_4\}$ are included in $paths(D) - \{p.@l\}$. Thus, $(D', \Sigma') \nvdash S_3 \rightarrow S_4$.

To conclude the proof we note that $p.@l \in AP(D, \Sigma)$ and $p.@l \notin AP(D', \Sigma')$, since $p.@l \notin paths(D')$. Therefore, $AP(D', \Sigma') \subsetneq AP(D, \Sigma)$. $\square$

(1) If (D, Σ) is in XNF then return (D, Σ), otherwise go to step (2).
(2) If there is an anomalous FD $X \to p.@l$ and $q \in \mathit{EPaths}(D)$ such that $q \in X$ and $q \to X \in (D, \Sigma)^+$, then:
  (2.1) Choose a fresh attribute $@m$
  (2.2) $D := D[p.@l := q.@m]$
  (2.3) $\Sigma := \Sigma[p.@l := q.@m]$
  (2.4) Go to step (1)
(3) Choose a (D, Σ)-minimal anomalous FD $X \to p.@l$, where $X = \{q, p_1.@l_1, \ldots, p_n.@l_n\}$
  (3.1) Create fresh element types $\tau, \tau_1, \ldots, \tau_n$
  (3.2) $D := D[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$
  (3.3) $\Sigma := \Sigma[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$
  (3.4) Go to step (1)

Fig. 5. XNF decomposition algorithm.

6.1.3 *The algorithm.* The algorithm applies the two transformations presented in the previous sections until the schema is in XNF, as shown in figure 5. Step (2) of the algorithm corresponds to the "moving attributes" rule applied to an anomalous FD $q \to p.@l$ and step (3) corresponds to the "creating new element types" rule applied to an anomalous FD $\{q, p_1.@l_1, \ldots, p_n.@l_n\} \to p.@l$. We choose to apply first the "moving attributes" rule since the other one involves minimality testing .

The algorithm shows in figure 5 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 (Propositions 6.1 and 6.2), we obtain:

THEOREM 6.3. *The XNF decomposition algorithm terminates, and outputs a specification (D, Σ) in XNF.*

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 Propositions 6.1 and 6.2 yields:

PROPOSITION 6.4. *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.*

6.2 Lossless Decomposition

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 "calculously dominance" from [Hull 1986]. That notion requires the existence of two relational algebra queries that translate back and forth between two relational schemas. Adapting the definition of [Hull 1986] is problematic in our setting, as no XML query language yet has the same "yardstick" status as relational algebra for relational databases.

Instead, we define (D', Σ') as a lossless decomposition of (D, Σ) if there is a mapping f from paths in the DTD D' to paths in the DTD D such that for every tree $T \models (D, \Sigma)$, there is a tree $T' \models (D', \Sigma')$ such that T and T' agree on all the paths with respect to this mapping f.

This can be done formally using the relational representation of XML trees via the $\mathit{tuples}_D(\cdot)$ operator. Given DTDs D and D', a function $f : \mathit{paths}(D') \to \mathit{paths}(D)$ is a *mapping from D' to D* if f is onto and a path p is an element path in D' if and only if $f(p)$ is an element path in D. Given tree tuples $t \in \mathcal{T}(D)$ and $t' \in \mathcal{T}(D')$, we write $t \equiv_f t'$ if for all $p \in \mathit{paths}(D') - \mathit{EPaths}(D')$, $t'.p = t.f(p)$. Given nonempty sets of tree tuples $X \subseteq \mathcal{T}(D)$ and $X' \subseteq \mathcal{T}(D')$, we let $X \equiv_f X'$ if for every $t \in X$, there exists $t' \in X'$ such that $t \equiv_f t'$, and for every $t' \in X'$, there exist $t \in X$ such that $t \equiv_f t'$. Finally, if T and T' are XML trees such that $T \lhd D$ and $T' \lhd D'$, we write $T \equiv_f T'$ if $\mathit{tuples}_D(T) \equiv_f \mathit{tuples}_{D'}(T')$.

Definition 6.5. Given XML specifications (D, Σ) and (D', Σ'), (D', Σ') is a lossless decomposition of (D, Σ), written $(D, \Sigma) \leq_{\text{lossless}} (D', \Sigma')$, if there exists a mapping f from D' to D such that for every $T \models (D, \Sigma)$ there is $T' \models (D', \Sigma')$ such that $T \equiv_f T'$.

In other words, all information about a document conforming to (D, Σ) can be recovered from some document that conforms to (D', Σ').

It follows immediately from the definition that $\leq_{\text{lossless}}$ is transitive. Furthermore, we show that every step of the normalization algorithm is lossless.

PROPOSITION 6.6. *If (D', Σ') is obtained from (D, Σ) by using one of the transformations from the normalization algorithm, then $(D, \Sigma) \leq_{\text{lossless}} (D', \Sigma')$.*

PROOF. We consider the two steps of the normalization algorithm, and for each step generate a mapping f. The proofs that those mappings satisfy the conditions of Definition 6.5 are straightforward.

(1) Assume that the "moving attribute" transformation was used to generate (D', Σ'). Then $D' = D[p.@l := q.@m]$, $\Sigma' = \Sigma[p.@l := q.@m]$ and $q \to p.@l$ is an anomalous FD in $(D, \Sigma)^+$. In this case, the mapping f from D' to D is defined as follows. For every $p' \in \mathit{paths}(D') - \{q.@m\}$, $f(p') = p'$, and $f(q.@m) = p.@l$.
(2) Assume that the "creating new element types" transformation was used to generate (D', Σ'). Then (D', Σ') was generated by considering a (D, Σ)-minimal anomalous FD $\{q,\ p_1.@l_1,\ \ldots,\ p_n.@l_n\} \to p.@l$. Thus, $D' = D[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$ and $\Sigma' = \Sigma[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$. In this case, the mapping f from D' to D is defined as follows: $f(q.\tau) = p$, $f(q.\tau.@l) = p.@l$, $f(q.\tau.\tau_i) = p_i$, $f(q.\tau.\tau_i.@l_i) = p_i.@l_i$ and $f(p') = p'$ for the remaining paths $p' \in \mathit{paths}(D')$.

□

Thus, if (D', Σ') is the output of the normalization algorithm on (D, Σ), then $(D, \Sigma) \leq_{\text{lossless}} (D', \Sigma')$.

In relational databases, the definition of lossless decomposition indicates how to transform instances containing redundant information into databases without redundancy. This transformation uses the projection operator. Notice that Definition 6.5 also indicates a way of transforming XML documents to generate well-designed documents: If $(D, \Sigma) \leq_{\text{lossless}} (D', \Sigma')$, then for every $T \models (D, \Sigma)$ there exists $T' \models (D', \Sigma')$ such that T and T' contain the same data values. The mappings $T \mapsto T'$ corresponding to the two transformations of the normalization algorithm can be implemented in an XML query language, more precisely, using XQuery FLWOR[2] expressions. We use transformations of documents shown in Section 1 for illustration; the reader will easily generalize them to produce the general queries corresponding to the transformations of the normalization algorithm.

Example 6.7. Assume that the DBLP database is stored in a file `dblp.xml`. As shown in example 1.2, this document can contain redundant information since year is stored multiple times for a given conference. We can solve this problem by applying the "moving attribute" transformation and making year an attribute of issue. This transformation can be implementing by using the following FLWOR expression:

```
let $root := document("dblp.xml")/db
<db>
{ for $co in $root/conf
  <conf>
    <title> { $co/title/text() } </title>,
    { for $is in $co/issue
      let $value := $is/inproceedings[position() = 1]/@year
      <issue year="{ $value }">
      { for $in in $is/inproceedings
        <inproceedings key="{ $in/@key }" pages="{ $in/@pages }">
        { for $au in $in/author
          <author> { $au/text() } </author>,
          <title> { $in/title/text() } </title>
        }
        </inproceedings>
      }
      </issue>
    }
  </conf>
}
</db>
```

The XPath expression `$is/inproceedings[position() = 1]/@year` is used to retrieve for every issue the value of the attribute year in the first paper in that issue. For every issue this number is stored in a variable `$value` and it becomes the value of its attribute year: `<issue year="{ $value }">`. □

Example 6.8. Assume that the XML document shown in figure 1 is stored in a file `university.xml`. This document stores information about courses in a university and it contains redundant information since for every student taking a course we store his/her name. To solve this problem, we split the information about names and grades by creating an extra element type, `info`, for student information. This transformation can be implemented as follows.

[2]FLWOR stands for *for*, *let*, *where*, *order by*, and *return*.

```
let $root := document("university.xml")/courses
<courses>
{ for $co in $root/course
  <course> {-- Query that removes name as a child of student --} </course>,
  for $na in distinct-values($root/course/taken_by/student/name/text())
  <info>
  { for $nu in distinct-values($root/course/taken_by/student[name/text() =
                                                              $na]/@sno)
    <number sno="{ $nu }">,
    <name> { $na } </name>
  }
  </info>
}
</courses>
```

We omitted the query that removes `name` as a child of `student` since it can be done as in the previous example. □

6.3 Eliminating additional assumptions

Finally, we have to show how to get rid of the additional assumption that for every anomalous FD $X \rightarrow p.@l$, every time that $p.@l$ is not null, every path in X is not null. We illustrate this by a simple example.

Assume that D is the DTD shown in figure 6 (a). Every XML tree conforming to this DTD has as root an element of type r which has a child of type either A or B and an arbitrary number of elements of type C, each of them containing an attribute $@l$. Let Σ be the set of FDs $\{r.A \rightarrow r.C.@l\}$. Then, (D, Σ) is not in XNF since $(D, \Sigma) \not\vdash r.A \rightarrow r.C$.

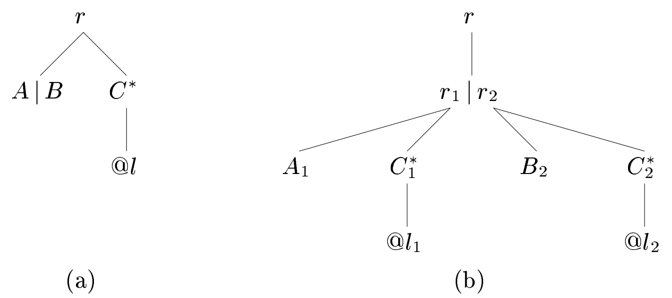


Fig. 6. Splitting a DTD.

If we want to eliminate the anomalous FD $r.A \rightarrow r.C.@l$, we cannot directly apply the algorithm presented in Section 6.1, since this FD does not satisfy the basic assumption made in that section; it could be the case that $r.C.@l$ is not null and $r.A$ is null. To solve this problem we transform (D, Σ) into a new XML specification (D', Σ') that is essentially equivalent to (D, Σ) and satisfies the assumption made in Section 6.1. The new XML specification is constructed by splitting the disjunction. More precisely, DTD D' is defined as the DTD shown in figure 6 (b). This DTD contains two copies of the DTD D, one of then containing element type A, denoted by A_1, and the other one containing element type B, denoted by B_2. The set of functional dependencies Σ' is constructed by including the FD $r.A \to r.C.@l$ in both DTDs, that is, $\Sigma' = \{r.A_1 \to r.C_1.@l_1,\, r.A_2 \to r.C_2.@l_2\}$.

In the new specification (D', Σ'), the user chooses between having either A or B by choosing between either r_1 or r_2. We note that the new FD $r.A_2 \to r.C_2.@l_2$ is trivial and, therefore, to normalize the new specification we only have to take into account FD $r.A_1 \to r.C_1.@l_1$. This functional dependency satisfies the assumption made in Section 6.1, so we can use the decomposition algorithm presented in that section.

It is straightforward to generalize the methodology presented in the previous example for any DTD. In particular, if we have an arbitrary regular expression s in a DTD $D = (E, A, P, R, r)$ and we have to split it into one regular expression containing an element type $\tau \in E$ and another one not containing this symbol, we consider regular expressions $s \cap (E^* \tau E^*)$ and $s - (E^* \tau E^*)$.

7. REASONING ABOUT FUNCTIONAL DEPENDENCIES

In the previous section we saw that it is possible to losslessly convert a DTD into one in XNF. The algorithm used XML functional dependency implication. Although XML FDs and relational FDs are defined similarly, the implication problem for the former class is far more intricate. In this section we study the implication problem for XML functional dependencies. In sections 7.1 and 7.2 we introduce two classes of DTDs for which the implication problem can be solved efficiently. These classes include most of real-world DTDs. In section 7.3 we introduce two classes of DTDs for which the implication problem is coNP-complete. In section 7.4 we show that, unlike relational FDs, XML FDs are not finitely axiomatizable. Finally, in section 7.5 we study the complexity of the XNF satisfaction problem. In all these sections we assume, without loss of generality, that all FDs have a single path on the right-hand side.

7.1 Simple regular expressions

Typically, regular expressions used in DTDs are rather simple. We now formulate a criterion for simplicity that corresponds to a common 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. Simple regular expressions were also considered in [Abiteboul et al. 2001] under the name of *multiplicity atoms*.

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 [ebXML 2001], a set of specifications to conduct business over the Internet, is a simple DTD. Part of this schema is showed in figure 7.

THEOREM 7.1. *The implication problem for FDs over simple DTDs is solvable*

```
<!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*)>
```

Fig. 7. Part of the Business Process Specification Schema of ebXML.

in quadratic time.

PROOF SKETCH: Here we present the sketch of the proof. The complete proof can be found in electronic appendix A.1.

In the first part of the proof we show that given a simple DTD D and a set of FDs $\Sigma \cup \{S \to p\}$ over D, the problem of verifying whether $\Sigma \not\vdash S \to p$ can be reduced to the problem of finding a counterexample to a certain implication problem. That is, we need to find an XML tree T such that $T \models (D, \Sigma)$, $T \not\models S \to p$, T contains two tree tuples and T satisfies some additional conditions that depend on the simplicity of D. Essentially, if an element type is allowed to occur zero times ($a?$ or a^*), then in constructing the counterexample such elements not need to be considered if they are irrelevant to the functional dependencies under consideration. Furthermore, all the element types in a regular expression in D can be considered independently. Observe that this condition is not longer valid if a regular expression in D contains a disjunction (D is not simple). For instance, if $(a|b)$ is a regular expression in D, then a and b are not independent; if a does not appear in an XML tree conforming to D, then b appears in this tree.

In the second part of the proof we show that the problem of finding this counterexample can be reduced to the problem of verifying if a certain propositional formula φ, constructed from D and $\Sigma \cup \{S \to p\}$, is satisfiable. This formula is of the form $\varphi_1 \vee \cdots \vee \varphi_n$, where n is at most the length of the path p and each φ_i ($i \in [1, n]$) is a conjunction of Horn clauses and is of linear size in the size of D and $\Sigma \cup \{S \to p\}$. Given that the consistency problem for Horn clauses is solvable in linear time [Dowling and Gallier 1984], we conclude that the counterexample can be found in quadratic time and, therefore, our original problem can be solved in quadratic time. □

7.2 Small number of disjunctions

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. For example, the following DTD cannot be represented as a simple DTD:

```
<!DOCTYPE university [
  <!ELEMENT university (course*)>
  <!ELEMENT course (number, student*)>
```

```
  <!ELEMENT number (#PCDATA)>
  <!ELEMENT student ((name | FLname), grade)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT FLname (first_name, last_name)>
  <!ELEMENT first_name (#PCDATA)>
  <!ELEMENT last_name (#PCDATA)>
  <!ELEMENT grade (#PCDATA)>
]>
```

In this example, every student must have a name. This name can be an string or it can be a composition of a first and a last name. It is desirable to express constraints on this kind of DTDs. For instance,

$$student.name.\mathtt{S} \rightarrow student,$$
$$\{student.FLname.first\_name.\mathtt{S},\, student.FLname.last\_name.\mathtt{S}\} \rightarrow student,$$

are functional dependencies in this domain. It is also desirable to reason about these constraints efficiently. Often, a DTD is not simple because a small number of regular expressions in it are not simple. In this section we will show that there is a polynomial time algorithm for reasoning about constraints over DTDs containing a small number of disjunctions.

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, $N_\tau = |\{p \in paths(D) \mid last(p) = \tau\}| \times N_{s_1} \times \cdots \times N_{s_n}$ otherwise. Finally, $N_D = \prod_{\tau \in E} N_\tau$.

THEOREM 7.2. *For any fixed $c > 0$, the FD implication problem for disjunctive DTDs D with $N_D \leq \|D\|^c$ is solvable in polynomial time*[3].

PROOF SKETCH: Here we present the sketch of the proof. The complete proof can be found in electronic appendix A.2.

The main idea of this proof is that the implication problem for disjunctive DTDs can be reduced to a number of implication problems for simple DTDs by splitting the disjunctions. More precisely, given a disjunctive DTD D and a set of functional dependencies $\Sigma \cup \{S \rightarrow p\}$ over D, there exist (D_1, Σ_1), …, (D_n, Σ_n) such that each D_i ($i \in [1,n]$) is a simple DTD, Σ_i is a set of functional dependencies over D_i ($i \in [1,n]$) and $(D, \Sigma) \vdash S \rightarrow p$ if and only if $(D_i, \Sigma_i) \vdash S \rightarrow p$ for every $i \in [1,n]$. The number n of implication problems for simple DTDs is at most N_D. Thus,

[3] $\|\cdot\|$ is the size of the description of an object. For instance, $\|p\|$ is the length of the path p and $\|S\|$ is the sum of the lengths of the paths in S.

since the implication problem for simple DTDs can be solved in quadratic time (see Theorem 7.1), the implication problem for disjunctive DTDs D with $N_D \leq \|D\|^c$, for some constant c, can be solved in polynomial time. $\square$

7.3 Relational DTDs

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 [Higgins and Jelliffe 1999]:

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

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

PROPOSITION 7.3. *Every disjunctive DTD is relational.*

PROOF. Let $D = (E, A, P, R, r)$ be a disjunctive DTD, T an XML tree conforming to D and X a non-empty subset of $\mathit{tuples}_D(T)$. Assume that $\mathit{trees}_D(X) \not\models D$, that is, there is an XML tree $T' = (V, \mathit{lab}, \mathit{ele}, \mathit{att}, \mathit{root})$ in $\mathit{trees}_D(X)$ such that $T' \not\models D$. Then, there is a vertex $v \in V$ reachable from the root by following a path p such that $\mathit{lab}(v) = \tau$ and $\mathit{ele}(v)$ does not conform to the regular expression $P(\tau)$.

If $P(\tau) = s$, where s is a simple disjunction over an alphabet A, then there is $t' \in X$ such that $t'.p = v$ and $t'.p.a = \perp$, for each $a \in A$. Thus, given that $T \models D$, we conclude that there is a tuple $t \in \mathit{tuples}_D(T)$ such that $t.p.b \neq \perp$, for some $b \in A$, and $t'.w = t.w$ for each $w \in \mathit{paths}(D)$ such that $p.b$ is not a prefix of w. Hence, $t' \sqsubset t$. But, this contradicts the definition of $\mathit{tuples}_D(\cdot)$, since $t', t \in \mathit{tuples}_D(T)$. The proof for $P(\tau) = s_1, \ldots, s_n$, where each s_i ($i \in [1, n]$) is either a simple regular expression or a simple disjunction, is similar. $\square$

THEOREM 7.4. *The FD implication problem over relational DTDs and over disjunctive DTDs is coNP-complete.*

PROOF. Here we prove the intractability of the implication problem for disjunctive DTDs. The coNP membership proof can be found in electronic appendix A.3.

In order to prove the coNP-hardness, we will reduce SAT-CNF to the complement of the implication problem for disjunctive DTDs. Let θ be a propositional formula of the form $C_1 \wedge \cdots \wedge C_n$, where each C_i ($i \in [1, n]$) is a clause. Assume that θ uses propositional variables $x_1, \ldots, x_m$.

We need to construct a disjunctive DTD D and a set of functional dependencies $\Sigma \cup \{\varphi\}$ such that $(D, \Sigma) \not\vdash \varphi$ if and only if θ is satisfiable. We define the DTD $D = (E, A, P, R, r)$ as follows.

$$
\begin{aligned}
E &= \{r, B, C\} \cup \{C_{i,j} \mid C_i \text{ mentions literal } x_j\} \cup \{\bar{C}_{i,j} \mid C_i \text{ mentions literal } \neg x_j\}, \\
A &= \{@l\}.
\end{aligned}
$$

In order to define P, first we define a function for translating clauses into regular expressions. If the set of literal mentioned in the clause C_i ($i \in [1, n]$) is

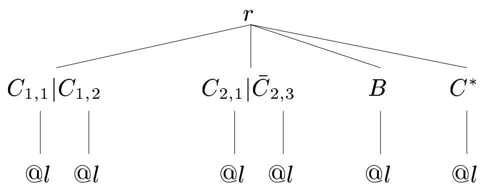


Fig. 8. DTD generated from a formula $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_3)$.

$\{x_{j_1}, \ldots, x_{j_p}, \bar{x}_{k_1}, \ldots, \bar{x}_{k_q}\}$, then

$$tr(C_i) \ = \ C_{i,j_1}|\cdots|C_{i,j_p}|\bar{C}_{i,k_1}|\cdots|\bar{C}_{i,k_q}.$$

We define the function P on the root as $P(r) = tr(C_1), \ldots, tr(C_n), B, C^*$. For the remaining elements of E, we define P as ϵ. Finally, $R(r) = \emptyset$ and $R(\tau) = \{@l\}$ for every $\tau \in E - \{r\}$. For example, figure 8 shows the DTD generated from a propositional formula $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_3)$.

For each pair of elements $C_{i,j}, \bar{C}_{k,j} \in E$, the set of functional dependencies Σ includes the constraint $\{r.C_{i,j}.@l,\ r.\bar{C}_{k,j}.@l\} \rightarrow r.C.@l$. Functional dependency φ is defined as $r.B.@l \rightarrow r.C.@l$.

We now prove that $(D, \Sigma) \not\vdash \varphi$ if and only if θ is satisfiable.

($\Rightarrow$) Suppose that $(D, \Sigma) \not\vdash \varphi$. Then, there is an XML tree T such that $T \models (D, \Sigma)$ and $T \not\models \varphi$. We define a truth assignment σ from T as follows. For each $j \in [1, m]$, if there is $i \in [1, n]$ such that r has a child of type $C_{i,j}$ in T, then $\sigma(x_j) = 1$, otherwise $\sigma(x_j) = 0$. We now prove that $\sigma \models C_i$, for each $i \in [1, n]$. By definition of D, there is $j \in [1, m]$ such that r has a child in T of type either $C_{i,j}$ or $\bar{C}_{i,j}$. In the first case, C_i contains the literal x_j and $\sigma(x_j) = 1$, by definition of σ. Therefore, $\sigma \models C_i$. In the second case, C_i contains a literal $\neg x_j$. If $\sigma(x_j) = 1$, then there is $k \in [1, n]$ such that r has a child of type $C_{k,j}$ in T, by definition of σ. Since $\{r.C_{k,j}.@l,\ r.\bar{C}_{i,j}.@l\} \rightarrow r.C.@l$ is a constraint in Σ, all the nodes in T of type C have the same value in the attribute $@l$. Thus, $T \models r.B.@l \rightarrow r.C.@l$, a contradiction. Hence, $\sigma(x_j) = 0$ and $\sigma \models C_i$.

($\Leftarrow$) Suppose that θ is satisfiable. Then, there exists a truth assignment σ such that $\sigma \models \theta$. We define an XML tree T conforming to D as follows. For each $i \in [1, n]$, choose a literal l_j in C_i such that $\sigma \models l_j$. If $l_j = x_j$, then r has a child of type $C_{i,j}$ in T, otherwise r has a child of type $\bar{C}_{i,j}$ in T. Moreover, r has one child of type B and two children of type C. We assign two distinct values to the attribute $@l$ of the nodes of type C, and the same value to the rest of the attributes in T. Thus, $T \not\models \varphi$, and it is easy to verify that $T \models \Sigma$. This completes the proof. $\square$

7.4 Nonaxiomatizability of XML functional dependencies

In this section we present a simple proof that XML FDs cannot be finitely axiomatized. This proof shows that, unlike relational databases, there is a nontrivial interaction between DTDs and functional dependencies. To present this proof we need to introduce some terminology.

Given a DTD D and a set of functional dependencies Σ over D, we say that (D, Σ) is *closed under implication* if for every FD φ over D such that $(D, \Sigma) \vdash \varphi$, it is the case that $\varphi \in \Sigma$. Furthermore, we say that (D, Σ) is *closed under k-ary implication*, $k \geq 0$, if for every FD φ over D, if there exists $\Sigma' \subseteq \Sigma$ such that $|\Sigma'| \leq k$ and $(D, \Sigma') \vdash \varphi$, then $\varphi \in \Sigma$. For example, if (D, Σ) is closed under 0-ary implication, then Σ contains all the trivial FDs.

Since the implication problem for relational FDs is finitely axiomatizable, there exists $k \geq 0$ such that each relation schema $R(A_1, \ldots, A_n)$ admits a k-ary ground axiomatization for the implication problem, that is, an axiomatization containing rules of the form *if* Γ *then* γ, where $\Gamma \cup \{\gamma\}$ is a set of FDs over $R(A_1, \ldots, A_n)$ and $|\Gamma| \leq k$. For instance, $R(A, B, C)$ admits a 2-ary ground axiomatization including, among others, the following rules: *if* $\emptyset$ *then* $AB \to A$, *if* $A \to B$ *then* $AC \to BC$ and *if* $\{A \to B, B \to C\}$ *then* $A \to C$. Similarly, if there exists a finite axiomatization for the implication problem of XML FDs, then there exists $k \geq 0$ such that each DTD D admits a (possible infinite) k-ary ground axiomatization for the implication problem. The contrapositive of the following proposition gives us a sufficient condition for showing that the XML FD implication problem does not admit a k-ary ground axiomatization for every $k \geq 0$ and, therefore, it does not admit a finite axiomatization.

PROPOSITION 7.5. *For every $k \geq 0$, if there is a k-ary ground axiomatization for the implication problem of XML FDs, then for every DTD D and set of FDs Σ over D, if (D, Σ) is closed under k-ary implication then (D, Σ) is closed under implication.*

PROOF. This proposition was proved in [Abiteboul et al. 1995] for the case of relational databases. The proof for XML FDs is similar. ☐

THEOREM 7.6. *The implication problem for XML functional dependencies is not finitely axiomatizable.*

PROOF. By Proposition 7.5, for every $k \geq 0$ we need to exhibit a DTD D_k and a set of functional dependencies Σ_k such that (D_k, Σ_k) is closed under k-ary implication and (D_k, Σ_k) is not closed under implication.

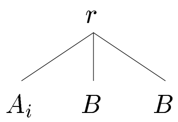
The DTD $D_k = (E, A, P, R, r)$ is defined as follows: $E = \{A_1, \ldots, A_k, A_{k+1}, B\}$, $A = \emptyset$, $P(r) = (A_1 | \cdots | A_k | A_{k+1}), B^*$ and $P(\tau) = \epsilon$ for every $\tau \in E - \{r\}$. The set of FDs Σ_k is defined as the union of the following sets:

—$\{r.A_i \to r.B \mid i \in [1, k+1]\} \cup \{\{r, r.A_i\} \to r.B \mid i \in [1, k+1]\}$,

—$\{S \to p \mid S \to p$ is a trivial FD in $D_k\}$.

It is easy to see that if φ is not a trivial functional dependency in D_k and $\varphi \notin \Sigma_k$, then $\varphi = r \to r.B$. Thus, in order to prove that (D_k, Σ_k) is closed under k-ary implication and is not closed under implication, we have to show that:

(1) For every $\Sigma' \subseteq \Sigma_k$ such that $|\Sigma'| \leq k$, $(D_k, \Sigma') \nvdash r \to r.B$. Since $|\Sigma'| \leq k$, there exists $i \in [1, k+1]$ such that $r.A_i \to r.B \notin \Sigma'$ and $\{r, r.A_i\} \to r.B \notin \Sigma'$. Thus, an XML tree T defined as



conforms to D_k, satisfies Σ' and does not satisfy $r \to r.B$. We conclude that $(D_k, \Sigma') \not\vdash r \to r.B$.

(2) $(D_k, \Sigma_k) \vdash r \to r.B$. This proof is straightforward.

This completes the proof of the theorem. □

7.5 The complexity of testing XNF

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

PROPOSITION 7.7. *Given a relational DTD D and a set Σ of FDs over D, (D, Σ) is in XNF iff for each nontrivial FD of the form $S \to p.@l$ or $S \to p.\mathtt{S}$ in Σ, $S \to p \in (D, \Sigma)^+$.*

PROOF. The proof is given in electronic appendix A.4. □

From this, we immediately derive:

COROLLARY 7.8. *Testing if (D, Σ) is in XNF can be done in cubic time for simple DTDs, and is coNP-complete for relational DTDs.*

8. RELATED WORK AND FUTURE RESEARCH

It was introduced in [Embley and Mok 2001] an XML normal form defined in terms of functional dependencies, multi-valued dependencies and inclusion constraints. Although that normal form was also called XNF the approach of [Embley and Mok 2001] was very different from ours. The normal form of [Embley and Mok 2001] was defined in terms of two conditions: XML specifications must not contain redundant information with respect to a set of constraints, and the number of schema trees (see Section 5.2) must be minimal. The normalization process is similar to the ER approach in relational databases. A conceptual-model hypergraph is constructed to model the real world and an algorithm produces an XML specification in XNF. It was proved in [Arenas and Libkin 2003] that an XML specification given by a DTD D and a set Σ of XML functional dependencies is in XNF if and only if no XML tree conforming to D and satisfying Σ contains redundant information. Thus, for the class of functional dependencies defined in this paper, the XML normal form introduced in [Embley and Mok 2001] is more restrictive than our XML normal form.

Normal forms for extended context-free grammars, similar to the Greibach normal form for CFGs, were considered in [Albert et al. 2001]. These, however, do not necessarily guarantee good XML design.

The functional dependency language used in [Embley and Mok 2001] is based on a language for nested relations and it does not consider relative constraints. In a very recent paper [Lee et al. 2002] was introduced a language for expressing functional dependencies for XML. In that language, a functional dependency is an expression of the form $(p, [q_1, \ldots, q_n \to q_{n+1}])$, where p is a path and every q_i

($i \in [1, n+1]$) is of the form $\tau.@l$, where τ is an element type. An XML tree T satisfies this constraint if for any two subtrees T_1, T_2 of T whose roots are nodes reachable from the root of T by following path p, if T_1 and T_2 agree on the value of q_i, for every $i \in [1, n]$, then they agree on the value of q_{n+1}. This language does not consider relative constraints and its semantics only works properly if some syntactic restrictions are imposed on the functional dependencies [Lee et al. 2002]. The normalization problem is not considered in [Lee et al. 2002].

Other proposals for XML constraints (mostly keys) have been studied in [Buneman et al. 2001a; 2001b; Fan and Siméon 2000]; these constraints do not use DTDs. XML constraints that takes DTDs into account are studied in [Fan and Libkin 2001].

Numerous surveys of relational normalization can be found in the literature [Beeri et al. 1978; Kanellakis 1990; Abiteboul et al. 1995]. Normalization for nested relations and object-oriented databases is studied in [Özsoyoglu and Yuan 1987; Mok et al. 1996; Tari et al. 1997]. Coding nested relations into flat ones, similar to our tree tuples, is done in [Suciu 1997; Van den Bussche 2001]. We use functional dependencies over incomplete relations using the techniques from [Atzeni and Morfuni 1984; Buneman et al. 1991; Grahne 1991; Imielinski and Jr. 1984; Levene and Loizou 1998].

8.1 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 multi-valued dependencies which are naturally induced by the tree structure.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: `http://www.acm.org/pubs/citations/journals/tods/20YY-V-N/p1-URLend`.

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

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

Abiteboul, S., Segoufin, L., and Vianu, V. 2001. Representing and Querying XML with Incomplete Information. In *Proceedings of the Twentieth ACM Symposium on Principles of Database Systems.* 150 – 161.

Albert, J., Giammarresi, D., and Wood, D. 2001. Normal Form Algorithms for Extended Context-free Grammars. *Theoretical Computer Science 267,* 1-2, 35 – 47.

ARENAS, M. AND LIBKIN, L. 2003. An Information-Theoretic Approach to Normal Forms for Relational and XML Data. In *Proceedings of the Twenty-second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* 15 – 26.

ATZENI, P. AND MORFUNI, N. 1984. Functional Dependencies in Relations with Null Values. *Information Porcessing Letters 18,* 4, 233–238.

BEERI, C., BERNSTEIN, P., AND GOODMAN, N. 1978. A Sophisticate's Introduction to Database Normalization Theory. In *Fourth International Conference on Very Large Data Bases.* 113–124.

BUNEMAN, P., DAVIDSON, S., FAN, W., HARA, C., AND TAN, W. C. 2001a. Keys for XML. In *Proceedings of the Tenth International World Wide Web Conference.* 201–210.

BUNEMAN, P., DAVIDSON, S., FAN, W., HARA, C., AND TAN, W. C. 2001b. Reasoning about Keys for XML. In *Proceedings of the Eighth International Workshop on Database Programming Languages.*

BUNEMAN, P., JUNG, A., AND OHORI, A. 1991. Using Powerdomains to Generalize Relational Databases. *Theoretical Computer Science 91,* 1, 23–55.

DOWLING, W. AND GALLIER, J. 1984. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming 1,* 3, 267–284.

EBXML. 2001. Business Process Specification Schema v1.01. `http://www.ebxml.org/specs/`.

EMBLEY, D. AND MOK, W. Y. 2001. Developing XML Documents with Guaranteed "Good" Properties. In *Proceedings of the Twentieth International Conference on Conceptual Modeling.* 426–441.

FAN, W. AND LIBKIN, L. 2001. On XML Integrity Constraints in the Presence of DTDs. In *Proceedings of the Twentieth ACM Symposium on Principles of Database Systems.* 114 – 125.

FAN, W. AND SIMÉON, J. 2000. Integrity Constraints for XML. In *Proceedings of the Nineteenth ACM Symposium on Principles of Database Systems.* 23–34.

FLORESCU, D. AND KOSSMANN, D. 1999. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin 22,* 3, 27–34.

GRAHNE, G. 1991. *The Problem of Incomplete Information in Relational Databases.* Springer.

GUNTER, C. 1992. *Semantics of Programming Languages: Structures and Techniques.* MIT Press.

HIGGINS, J. AND JELLIFFE, R. 1999. QAML Version 2.4. `http://xml.ascc.net/resource/qaml-xml.dtd`.

HULL, R. 1986. Relative Information Capacity of Simple Relational Database Schemata. *SIAM Journal on Computing 15,* 3, 856–886.

IMIELINSKI, T. AND JR., W. L. 1984. Incomplete Information in Relational Databases. *Journal of the ACM 31,* 4, 761–791.

KANELLAKIS, P. 1990. Elements of Relational Database Theory. In *Handbook of Theoretical Computer Science*, Volume B, pages 1075-1144, MIT Press.

KANNE, C.-C. AND MOERKOTTE, G. 2000. Efficient Storage of XML Data. In *Proceedings of the 16th International Conference on Data Engineering.* 198.

LEE, M.-L., LING, T. W., AND LOW, W. L. 2002. Designing Functional Dependencies for XML. In *Proceedings of the Eighth International Conference on Extending Database Technology.* 124–141.

LEVENE, M. AND LOIZOU, G. 1998. Axiomatisation of Functional Dependencies in Incomplete Relations. *Theoretical Computer Science 206,* 1-2, 283–300.

LEY, M. 2003. DBLP. `http://www.informatik.uni-trier.de/~ley/db/index.html`.

MOK, W. Y., NG, Y.-K., AND EMBLEY, D. 1996. A Normal Form for Precisely Characterizing Redundancy in Nested Relations. *ACM Transactions on Database Systems 21,* 1, 77–106.

ÖZSOYOGLU, M. AND YUAN, L.-Y. 1987. A New Normal Form for Nested Relations. *ACM Transactions on Database Systems 12,* 1, 111–136.

SAGIV, Y., DELOBEL, C., PARKER, D. S., AND FAGIN, R. 1981. An Equivalence Between Relational Database Dependencies and a Fragment of Propositional Logic. *Jornal of the ACM 28,* 3, 435–453.

Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D., and Naughton, J. 1999. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases*. 302–314.

Suciu, D. 1997. Bounded Fixpoints for Complex Objects. *Theoretical Computer Science 176,* 1-2, 283–328.

Tari, Z., Stokes, J., and Spaccapietra, S. 1997. Object Normal Forms and Dependency Constraints for Object-Oriented Schemata. *ACM Transactions on Database Systems 22,* 4, 513–569.

Tatarinov, I., Ives, Z., Halevy, A., and Weld, D. 2001. Updating XML. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. 413–424.

Van den Bussche, J. 2001. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science 254,* 1-2, 363–377.

W3C. 1998. XML-Data, W3C Note. `http://www.microsoft.com/standards/xml/xmldata-f.htm`.

W3C. 2001. XML Schema, W3C Working Draft. `http://www.w3.org/XML/Schema`.

Received Month Year; revised Month Year; accepted Month Year

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

A Normal Form for XML Documents

MARCELO ARENAS
University of Toronto
and
LEONID LIBKIN
University of Toronto

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY, Pages 1–37.

A. PROOF OF SECTION 7

A DTD D can be inconsistent in the sense that there is no XML tree T such that $T \models D$. For example, a recursive DTD containing a rule $P(a) = a$ is not consistent; there is no a finite XML tree satisfying this rule. In this section we only consider consistent DTDs, since the implication problem for inconsistent DTDs is trivial and it can be checked in linear time whether a DTD is consistent [Fan and Libkin 2001].

A.1 Proof of Theorem 7.1

To prove this theorem we start by introducing some terminology. Given a simple DTD $D = (E, A, P, R, r)$ and $p, p' \in paths(D)$ such that p is a proper prefix of p', we say that p' *can be nullified from* p if p' is of the form $p.w_1.\cdots.w_n$, where $w_i \in E \cup A \cup \{\mathsf{S}\}$ ($i \in [1, n]$) and either (1) $P(last(p))$ contains $w_1?$ or w_1^*; or (2) there is $i \in [1, n-1]$ such that $P(w_i)$ contains $w_{i+1}?$ or w_{i+1}^*. Intuitively, p' can be nullified from p if there exists and XML tree T conforming to D and a tree tuple t in T such that $t.p \neq \perp$ and $t.p' = \perp$. For example, if $P(r) = a$, $P(a) = b^*$ and $P(b) = c$, then $r.a.b.c$ can be nullified from r and $r.a$, but it cannot be nullified from $r.a.b$. Given $S \subseteq paths(D)$, we say that p' *can be nullified from* S if p' can be nullified from p, where p is the longest common prefix of p' and a path from S.

The following is proved by the same argument as Lemma A.6 shown in electronic appendix A.3.

LEMMA A.1. *Given a simple DTD D, a set Σ of functional dependencies over D and $S \cup \{p\} \subseteq paths(D)$, $(D, \Sigma) \not\vdash S \to p$ if and only if there is an XML tree T and a path q prefix of p such that $T \models (D, \Sigma)$, $tuples_D(T) = \{t_1, t_2\}$, $t_1.S = t_2.S$, $t_1.S \neq \perp$, $t_1.p \neq t_2.p$, $t_1.p \neq \perp$, $t_2.p \neq \perp$, $t_1.q \neq t_2.q$ and*

—*For each $s \in paths(D)$, if s can be nullified from $S \cup \{p\}$, then $t_1.s = t_2.s = \perp$.*

—*For each* $s \in paths(D)$*, if* q *is not a prefix of* s *and* s *cannot be nullified from* $S \cup \{p\}$*, then* $t_1.s = t_2.s$ *and* $t_1.s \neq \perp$*.*

To prove that the implication problem for simple DTDs can be solved in polynomial time, we use the technique of [Sagiv et al. 1981] and code constraints with propositional formulas. That is, for each simple DTD D and set of functional dependencies $\Sigma \cup \{S \to p\}$ over D, we will define a propositional formula φ such that $(D, \Sigma) \not\vdash S \to p$ if and only if φ is satisfiable. This formula will be of the form $\varphi_1 \vee \cdots \vee \varphi_n$, where each φ_i ($i \in [1, n]$) is a conjunction of Horn clauses. Given that the consistency problem for Horn clauses is solvable in linear time, we will conclude that our problem is solvable in quadratic time.

Let D be a DTD, Σ a set of functional dependencies over D and $S \cup \{p\} \subseteq paths(D)$. Recall that we assumed that each constraints in Σ is of the form $S' \to p'$, where $S' \cup \{p'\} \subseteq paths(D)$. We define $paths(\Sigma)$ as $\{s \mid$ there is $S' \to p' \in \Sigma$ such that $s \in S' \cup \{p'\}\}$. To define the propositional formula φ we view each path $s \in paths(\Sigma) \cup S \cup \{p\}$ as a propositional variable. Furthermore, for each path q which is a prefix of p we define a propositional formula φ_q as

$$\neg p \wedge (\bigwedge_{s \in P_q \cup S} s) \wedge (\bigwedge_{s \in N_q} \neg s) \wedge \bigwedge_{\psi \in \Gamma} \psi,$$

where P_q, N_q and Γ are set of propositional variables and formulas defined as follows.

—For each $s \in paths(\Sigma)$ such that s cannot be nullified from $S \cup \{p\}$ and q is not a prefix of s, s is included in P_q.

—For each $s \in paths(\Sigma)$ such that $s \in EPaths(D)$, s cannot be nullified from $S \cup \{p\}$ and q is a prefix of s, s is included in N_q.

—For each $S' \to p' \in \Sigma$, if there is no $q' \in S' \cup \{p'\}$ such that q' can be nullified from $S \cup \{p\}$, then $(\bigwedge_{s \in S'} s) \to p'$ is included in Γ

We note that φ_q is a conjunction of Horn clauses.

The propositional formula φ is defined as the disjunction of some of the formula φ_qs. The following lemma shows that in this disjunction we only need to consider qs such that $q = q'.\tau$, for some $\tau \in E$, and $P(last(q'))$ contains τ^* or τ^+.

Lemma A.2. *Let* $D = (E, A, P, R, r)$ *be a simple DTD,* Σ *a set of functional dependencies over* D *and* $S \cup \{p, q\} \subseteq paths(D)$ *such that* q *is a prefix of* p*. If there is* $\tau \in E$ *such that* $q = q'.\tau$ *and* $P(last(q'))$ *contains* τ^* *or* τ^+*, then* φ_q *is satisfiable iff there is an XML tree* T *such that* $T \models (D, \Sigma)$*,* $tuples_D(T) = \{t_1, t_2\}$*,* $t_1.S = t_2.S$*,* $t_1.S \neq \perp$*,* $t_1.p \neq t_2.p$*,* $t_1.p \neq \perp$*,* $t_2.p \neq \perp$*,* $t_1.q \neq t_2.q$ *and*

—*For each* $s \in paths(D)$*, if* s *can be nullified from* $S \cup \{p\}$*, then* $t_1.s = t_2.s = \perp$*.*

—*For each* $s \in paths(D)$*, if* q *is not a prefix of* s *and* s *cannot be nullified from* $S \cup \{p\}$*, then* $t_1.s = t_2.s$ *and* $t_1.s \neq \perp$*.*

Proof. ($\Rightarrow$) Let σ be a truth assignment satisfying φ_q. We define tuples t_1 and t_2 as follows. For each $s \in paths(D)$, if s can be nullified from $S \cup \{p\}$, then $t_1.s = t_2.s = \perp$. If s cannot be nullified from $S \cup \{p\}$ we consider two cases. If q is not a prefix of s, then $t_1.s = t_2.s$ and $t_1.s \neq \perp$. Otherwise, if $\sigma(s) = 1$, then $t_1.s = t_2.s$ and $t_1.s \neq \perp$, else $t_1.s \neq t_2.s$, $t_1.s \neq \perp$ and $t_2.s \neq \perp$.

It is straightforward to prove that there is an XML tree $T \in \mathit{trees}_D(\{t_1, t_2\})$ such that $T \models D$ and $\mathit{tuples}_D(T) = \{t_1, t_2\}$. Given that $\sigma \models \neg p \wedge \bigwedge_{s \in S} s$, $t_1.S = t_2.S$, $t_1.S \neq \perp$, $t_1.p \neq t_2.p$, $t_1.p \neq \perp$ and $t_2.p \neq \perp$. Besides, $t_1.q \neq t_2.q$, since $q \in N_q$ and $\sigma \models \bigwedge_{s \in N_q} \neg s$. Thus, to finish the proof we have to show that $T \models \Sigma$. Let $S' \to p' \in \Sigma$. If there is $q' \in S' \cup \{p'\}$ such that q' can be nullified from $S \cup \{p\}$, then T trivially satisfies $S' \to p'$ since $t_1.q' = t_2.q' = \perp$. Otherwise, suppose that $t_1.S' = t_2.S'$ and $t_1.S' \neq \perp$. Then, by considering that $\sigma \models \bigwedge_{s \in P_q} s$ and the definition of t_1 and t_2, we conclude that $\sigma \models \bigwedge_{s \in S'} s$. Thus, given that $\sigma \models (\bigwedge_{s \in S'} s) \to p'$, we conclude that $\sigma(p') = 1$, and, therefore, $t_1.p' = t_2.p'$.

($\Leftarrow$) Suppose that there is an XML tree T satisfying the conditions of the lemma. Define a truth assignment σ as follows. For each $s \in \mathit{paths}(\Sigma) \cup S \cup \{p\}$, if $t_1.s \neq t_2.s$ then $\sigma(s) = 0$. Otherwise, $\sigma(s) = 1$.

Given that $t_1.p \neq t_2.p$ and $t_1.S = t_2.S$, $\sigma(\neg p) = 1$ and $\sigma \models \bigwedge_{s \in S} s$. Let $s \in P_q$. By definition, s cannot be nullified from $S \cup \{p\}$ and q is not a prefix of s, and, therefore, $t_1.s = t_2.s$. Thus, $\sigma(s) = 1$. We conclude that $\sigma \models \bigwedge_{s \in P_q} s$. Let $s \in N_q$. By definition, s cannot be nullified from $S \cup \{p\}$, q is a prefix of s and $s \in \mathit{EPaths}(D)$. Hence, $t_1.s \neq t_2.s$ and $\sigma(s) = 0$. We conclude that $\sigma \models \bigwedge_{s \in N_q} \neg s$. Finally, let $(\bigwedge_{s \in S'} s) \to p' \in \Sigma_q$. If $\sigma \models \bigwedge_{s \in S'} s$, then by definition of σ and Σ_q, we conclude that $t_1.S' = t_2.S'$ and $t_1.S' \neq \perp$. Thus, given that $T \models \Sigma$, we conclude that $t_1.p' = t_2.p'$ and, therefore, $\sigma(p') = 1$. □

Combining Lemmas A.1 and A.2 we obtain:

LEMMA A.3. *Let $D = (E, A, P, R, r)$ be a simple DTD, Σ a set of functional dependencies over D and $S \cup \{p\} \subseteq \mathit{paths}(D)$. Assume that $X = \{q \in \mathit{paths}(D) \mid q$ is a prefix of p and there is $\tau \in E$ such that $q = q'.\tau$ and $P(\mathit{last}(q'))$ contains τ^* or $\tau^+\}$. Then, $(D, \Sigma) \nvdash S \to p$ iff $\varphi = \bigvee_{q \in X} \varphi_q$ is satisfiable.*

Finally, we are ready to show that for a simple DTD D and a set of FDs $\Sigma \cup \{S \to p\}$ over D, checking whether $(D, \Sigma) \vdash S \to p$ can be done in quadratic time. The size of each formula φ_q in the previous Lemma is $O(\|\Sigma\| + \|S\| + \|p\|)$. Thus, it is possible to verify whether φ_q is satisfiable in time $O(\|\Sigma\| + \|S\| + \|p\|)$, since satisfiability of propositional Horn formulas can be checked in linear time [Dowling and Gallier 1984]. Hence, given that there are at most $\|p\|$ of these formulas, checking whether formula $\bigvee_{q \in X} \varphi_q$ in Lemma A.3 is satisfiable requires time $O(\|p\| \cdot (\|\Sigma\| + \|S\| + \|p\|))$. To construct this formula, first we execute two steps:

(1) For every $s \in \mathit{paths}(\Sigma)$, find the longest common prefix of s and a path from $S \cup \{p\}$, which requires time $O(\|s\| \cdot (\|S\| + \|p\|))$. By using this prefix verify whether s can be nullified from $S \cup \{p\}$, which requires time $O(\|s\| \cdot \|D\|)$.
(2) For each $s \in \mathit{paths}(\Sigma)$ and for each prefix q of p, verify whether q is a prefix of s, which requires time $O(\|q\|)$.

The total time required by these steps is $O(\|\Sigma\| \cdot (\|D\| + \|S\| + \|p\|))$. Let k be the number of paths in Σ and l be the number of prefixes of p. The information generated by the first step is stored in a array with k entries, one for each path in Σ, indicating whether each of these paths can be nullified from $S \cup \{p\}$. Similarly, the information generated by the second step is stored in l arrays with k entries each. By using these data structures, the formula $\bigvee_{q \in X} \varphi_q$ in Lemma A.3 can be constructed in time $O(\|p\| \cdot (\|\Sigma\| + \|S\| + \|p\|))$. Thus, the total time of the algorithm is $O(\|p\| \cdot (\|\Sigma\| + \|S\| + \|p\|) + \|\Sigma\| \cdot (\|D\| + \|S\| + \|p\|))$. This completes the proof of Theorem 7.1.

A.2 Proof of Theorem 7.2

To prove this theorem first we prove two lemmas. Let $D = (E, A, P, R, r)$ be a disjunctive DTD and $\tau \in E$ such that $P(\tau) = s_1, \ldots, s_n$. Assume that for a fixed $k \in [1, n]$, $s_k = s'_1 | s'_2$, where s'_1, s'_2 are simple disjunctions over alphabets A'_1, A'_2 and $A'_1 \cap A'_2 = \emptyset$. Assume that there is only one $p_\tau \in \mathit{paths}(D)$ such that $\mathit{last}(p_\tau) = \tau$. We define $\mathit{paths}_i(D)$ (for $i = 1, 2$) as the set of all paths q in D such that one of the following statement holds: (1) p_τ is not a proper prefix of q or (2) there is $\tau' \in E$ such that $p_\tau.\tau'$ is a prefix of q and τ' is in the alphabet of any of the regular expressions $s_1, \ldots, s_{k-1}, s'_i, s_{k+1}, \ldots, s_n$. Then we define DTDs $D_i = (E_i, A_i, P_i, R_i, r)$ (for $i = 1, 2$) as follows. $E_i = \{\tau' \in E \mid \tau'$ is mentioned in some $q \in \mathit{paths}_i(D)\}$, $A_i = \{@l \mid$ there is $\tau' \in E_i$ such that $@l \in R(\tau')\}$, $P_i(\tau) = s_1, \ldots, s_{k-1}, s'_i, s_{k+1}, \ldots, s_n$, $P_i(\tau') = P(\tau')$, for each $\tau' \in E_i - \{\tau\}$, and $R_i = R|_{E_i}$. Moreover, given a set of functional dependencies Σ over D, we define a set of functional dependencies Σ_i over D_i (for $i = 1, 2$) as follows. For each $S \to p \in \Sigma$, if $S \cup \{p\} \subseteq \mathit{paths}_i(D)$, then $S \to p$ is included in Σ_i.

LEMMA A.4. *Let D, Σ, τ, p_τ, D_i and Σ_i, for $i = 1, 2$ be as above and let $S \to p$ be a functional dependency over D. Then*

(a) If $S \cup \{p\} \not\subseteq \mathit{paths}_i(D)$ for every $i \in [1, 2]$, then $(D, \Sigma) \vdash S \to p$.

(b) If $S \cup \{p\} \subseteq \mathit{paths}_1(D)$ and $S \cup \{p\} \not\subseteq \mathit{paths}_2(D)$, then $(D, \Sigma) \vdash S \to p$ iff $(D_1, \Sigma_1) \vdash S \to p$.

(c) If $S \cup \{p\} \subseteq \mathit{paths}_i(D)$ for every $i \in [1, 2]$, then $(D, \Sigma) \vdash S \to p$ iff for every $i \in [1, 2]$, $(D_i, \Sigma_i) \vdash S \to p$.

PROOF. (a) Let $p_i \in \mathit{paths}_i(D)$ ($i \in [1, 2]$) such that $p_i \in S \cup \{p\}$, for every $i \in [1, 2]$, $p_1 \notin \mathit{paths}_2(D)$ and $p_2 \notin \mathit{paths}_1(D)$. Let T be an XML tree such that $T \models (D, \Sigma)$, and $t_1, t_2 \in \mathit{tuples}_D(T)$. Without loss of generality, assume that $p_1 \in S$. If $t_1.p_1 = t_2.p_1$ and $t_1.p_1 \neq \perp$, then $t_1.p_2 = t_2.p_2 = \perp$, and, therefore, $T \models S \to p$. Thus, we conclude that $(D, \Sigma) \vdash S \to p$.

(b) If $(D, \Sigma) \vdash S \to p$, we have to prove that $(D_1, \Sigma_1) \vdash S \to p$. Let T_1 be an XML such that $T_1 \models (D_1, \Sigma_1)$. This tree conforms to D and satisfies Σ, since each constraint $\varphi \in \Sigma - \Sigma_1$ contains at least one path q such that for every $t \in \mathit{tuples}_D(T_1)$, $t.q = \perp$. Hence, $T_1 \models S \to p$.

Suppose that $(D_1, \Sigma_1) \vdash S \to p$. We have to prove that $(D, \Sigma) \vdash S \to p$. Let T be an XML tree such that $T \models (D, \Sigma)$, and $t_1, t_2 \in \mathit{tuples}_D(T)$. Let $p_1 \in \mathit{paths}_1(D)$ such that $p_1 \in S \cup \{p\}$ and $p_1 \notin \mathit{paths}_2(D)$. By contradiction, suppose that $t_1.S = t_2.S$, $t_1.S \neq \perp$ and $t_1.p \neq t_2.p$. If $p_1 \in S$, then there is $T_1 \in \mathit{trees}_D(\{t_1, t_2\})$ such that $T_1 \models D_1$, since $t_1.p_1 \neq \perp$ and $t_2.p_1 \neq \perp$. Since $T \models \Sigma$, $T_1 \models \Sigma_1$, and, therefore $(D_1, \Sigma_1) \not\vdash S \to p$, a contradiction. If $p_1 = p$, without loss of generality, we can assume that $t_1.p_1 \neq \perp$. If $t_2.p_1 \neq \perp$, then there is $T_1 \in \mathit{trees}_D(\{t_1, t_2\})$ such that $T_1 \models D_1$. But, $T_1 \models \Sigma_1$, since $T \models \Sigma$, and, therefore $(D_1, \Sigma_1) \not\vdash S \to p$, a contradiction. Assume that $t_2.p_1 = \bot$. Define $t_2' \in \mathcal{T}(D_1)$ as follows. For each $w \in paths_1(D) \cap paths_2(D)$, $t_2'.w = t_2.w$, and for each $w \in paths_1(D) - paths_2(D)$, if $t_1.w = \bot$, then $t_2'.w = \bot$, otherwise $t_2'.w \neq t_1.w$. Given that $t_1.p_\tau \neq t_2.p_\tau$, since $t_1.p_1 \neq \bot$ and $t_2.p_1 = \bot$, we conclude that there is an XML tree $T_1 \in trees_D(\{t_1, t_2'\})$ such that T_1 conforms to D_1. But $T_1 \models \Sigma_1$, since $trees_D(\{t_1, t_2\}) \models \Sigma$. Thus, $(D_1, \Sigma_1) \not\vdash S \to p$, again a contradiction.

(c) We will only prove the "if" direction. The "only if" direction is analogous to the proof of this direction in (b). Assume that $(D, \Sigma) \not\vdash S \to p$. We will show that $(D_1, \Sigma_1) \not\vdash S \to p$ or $(D_2, \Sigma_2) \not\vdash S \to p$.

Given that every disjunctive DTD is a relational DTD (see Proposition 7.3), by Lemma A.6 we conclude that $(D, \Sigma) \not\vdash S \to p$ if and only if there is an XML tree T and a path q prefix of p such that $T \models (D, \Sigma)$, $tuples_D(T) = \{t_1, t_2\}$, $t_1.S = t_2.S$, $t_1.S \neq \bot$, $t_1.p \neq t_2.p$, $t_1.q \neq t_2.q$ and for each $s \in paths(D)$, if q is not a prefix of s, then $t_1.s = t_2.s$. We consider three cases.

(1) If q is not a prefix of p_τ. Then, there is $T' \in trees_D(\{t_1, t_2\})$ such that T' conforms to either D_1 or D_2. Without loss of generality, assume that $T' \models D_1$. In this case, $T' \models \Sigma_1$, since $T \models \Sigma$. Hence, $(D_1, \Sigma_1) \not\vdash S \to p$.

(2) If q is a prefix of p_τ and there exists $a_1' \in A_1'$ and $a_2' \in A_2'$ such that $t_1.p_\tau.a_1' \neq \bot$ and $t_2.p_\tau.a_2' \neq \bot$. In this case, we define $t_2' \in \mathcal{T}(D_1)$ as follows. For each $w \in paths_1(D) \cap paths_2(D)$, $t_2'.w = t_2.w$, and for each $w \in paths_1(D) - paths_2(D)$, if $t_1.w = \bot$, then $t_2'.w = \bot$, otherwise $t_2'.w \neq t_1.w$. Then, there exists $T' \in trees_{D_1}(\{t_1, t_2'\})$ such that $T' \models D_1$, $T' \models \Sigma_1$ and $T' \not\models S \to p$, since $T \models \Sigma$ and $T \not\models S \to p$. We conclude that $(D_1, \Sigma_1) \not\vdash S \to p$.

(3) If q is a prefix of p_τ and there are no $a_1' \in A_1'$ and $a_2' \in A_2'$ such that either $t_1.p_\tau.a_1' \neq \bot$ and $t_2.p_\tau.a_2' \neq \bot$ or $t_2.p_\tau.a_1' \neq \bot$ and $t_1.p_\tau.a_2' \neq \bot$. This case is analogous to the first one.

□

Given a disjunctive DTD $D = (E, A, P, R, r)$, to apply the previous lemma we need to find an element type τ such that there is exactly one path in D whose last element is τ and $P(\tau) = s_1, \ldots, s_k, \ldots, s_n$, where $s_k = s_1' | s_2'$, s_1' and s_2' are simple disjunctions over alphabets A_1', A_2' and $A_1' \cap A_2' = \emptyset$. If there is no such an element type and D is not a simple DTD, it is possible to create it by using the following transformation. Pick τ satisfying the previous conditions except for there is more than one path whose last element is τ. Pick $p \in paths(D)$ such that $last(p) = \tau$. Define a DTD $D_p = (E_p, A, P_p, R_p, r_p)$ as follows. $r_p = [r]$ and $E_p = (E - \{r\}) \cup \{[q] \mid q \in paths(D)$ and q is a prefix of $p\}$ (we use square brackets to distinguish between paths and element types). The functions P_p and R_p are defined as follows.

—For each $q \in paths(D)$ and $\tau' \in E$ such that $q.\tau'$ is a prefix of p, $P_p([q]) = f(P(last(q)))$, where f is a homomorphism defined as $f(\tau') = [q.\tau']$ and $f(\tau'') = \tau''$ for each $\tau'' \neq \tau'$. Moreover, $P_p([p]) = P(last(p))$ and $P_p(\tau') = P(\tau')$, for each $\tau' \in E - \{r\}$.

—For each $[q] \in E_p$, $R_p([q]) = R(last(q))$. Moreover, $R_p(\tau') = R(\tau')$, for each $\tau' \in E - \{r\}$.

Let $\Sigma \cup \{S \to q\}$ be a set of functional dependencies over D. We define a set of functional dependencies $\Sigma_p \cup \{S_p \to q_p\}$ over D_p as follows. For each path q' mentioned in $\Sigma \cup \{S \to q\}$, if $q' = q_1.q_2$, where q_1 is the longest common prefix of q' and p, then q' is replaced by $g(q_1).q_2$, where g is an homomorphism defined as $g([r]) = [r]$ and $g([w.\tau']) = g([w]).[w.\tau']$, for each $w.\tau'$ prefix of p. The following is straightforward.

Lemma A.5. *Let D, $\Sigma \cup \{S \to q\}$, D_p and $\Sigma_p \cup \{S_p \to q_p\}$ be as above. Then, $(D, \Sigma) \vdash S \to q$ iff $(D_p, \Sigma_p) \vdash S_p \to q_p$.*

Theorem 7.2 now follows from Lemmas A.4 and A.5.

A.3 The Implication Problem for Relational DTDs is in coNP

To prove this theorem we start with the following lemma.

Lemma A.6. *Given a relational DTD D, a set Σ of functional dependencies over D and $S \cup \{p\} \subseteq paths(D)$, $(D, \Sigma) \nvdash S \to p$ if and only if there is an XML tree T and a path q prefix of p such that T conforms to D, T satisfies Σ, $tuples_D(T) = \{t_1, t_2\}$, $t_1.S = t_2.S$, $t_1.S \neq \perp$, $t_1.p \neq t_2.p$, $t_1.q \neq t_2.q$ and for each $s \in paths(D)$, if q is not a prefix of s, then $t_1.s = t_2.s$.*

Proof. We will prove only the "only if" direction, since the "if" direction is trivial.

Suppose that $(D, \Sigma) \nvdash S \to p$. There is an XML tree T' conforming to D and satisfying Σ such that $T' \not\models S \to p$. Then, there are tuples $t'_1, t'_2 \in tuples_D(T)$ such that $t'_1.S = t'_2.S$, $t'_1.S \neq \perp$ and $t'_1.p \neq t'_2.p$. Let q be the shortest prefix of p such that $t'_1.q \neq t'_2.q$. We define tree tuples t_1 and t_2 as follows. For each $s \in paths(D)$, if q is not a prefix of s, then $t_1.s = t'_1.s$ and $t_2.s = t'_1.s$. Otherwise, $t_1.s = t'_1.s$ and $t_2.s = t'_2.s$. Notice that $t_1, t_2 \in tuples_D(T')$.

Given that D is a relational DTD, it is possible to find $T \in trees_D(\{t_1, t_2\})$ such that $T \models D$. We need to prove that T satisfies the conditions of the lemma. By definition of t_1 and t_2, $tuples_D(T) = \{t_1, t_2\}$ and for each $s \in paths(D)$, if q is not a prefix of s, then $t_1.s = t_2.s$. Besides, $t_1.S = t_2.S$, $t_1.S \neq \perp$ and $t_1.p \neq t_2.p$, since $t'_1.S = t'_2.S$, $t'_1.S \neq \perp$, $t'_1.p \neq t'_2.p$ and q is a prefix of p. Finally, $t_1.q \neq t_2.q$, since $t'_1.q \neq t'_2.q$, and $T \models \Sigma$, since $T' \models \Sigma$ and $t_1, t_2 \in tuples_D(T')$. $\square$

Now we are ready to prove that the implication problem for relational DTDs is in coNP. Let D be a relational DTD, Σ a set of functional dependencies over D and $S \cup \{p\} \subseteq paths(D)$. Let $prefix(\Sigma \cup \{S \to p\})$ be the set of all $p' \in paths(D)$ such that p' is a prefix of a path mentioned in $\Sigma \cup \{S \to p\}$. Notice that $\|prefix(\Sigma \cup \{S \to p\})\|$ is $O(\|\Sigma \cup \{S \to p\}\|^2)$.

To check whether $(D, \Sigma) \nvdash S \to p$, we use a nondeterministic algorithm that guesses the tuples t_1 and t_2 mentioned in Lemma A.6. This algorithm does not construct all the values in t_1 and t_2, it guesses only the values of these tuples that are necessary to verify whether $trees_D(\{t_1, t_2\}) \models \Sigma$. The algorithm works as follows. For each $s \in prefix(\Sigma \cup \{S \to p\})$, guess the values of $t_1.s$ and $t_2.s$. Verify whether it is possible to construct an XML tree conforming to D and containing

t_1 and t_2. If this does not hold, then return "no". Otherwise, guess a prefix q of p. Verify whether $t_1.S = t_2.S$, $t_1.S \neq \bot$, $t_1.p \neq t_2.p$, $t_1.q \neq t_2.q$ and for each $s \in paths(\Sigma \cup \{S \rightarrow p\})$, if q is not a prefix of s, then $t_1.s = t_2.s$. If this does not hold, then return "no". Otherwise, check whether the values in t_1 and t_2 satisfy Σ. If this is the case, then return "yes", otherwise return "no".

The previous algorithm works in nondeterministic polynomial time, since $\|prefix(\Sigma \cup \{S \rightarrow p\})\|$ is $O(\|\Sigma \cup \{S \rightarrow p\}\|^2)$. Therefore, we conclude that the implication problem for relational DTDs is in coNP.

A.4 Proof of Proposition 7.7

We only need to prove the "if" direction. Suppose that for each nontrivial FD of the form $S \rightarrow p.@l$ or $S \rightarrow p.\mathsf{S}$ in Σ, $S \rightarrow p \in (D, \Sigma)^+$.

Assume that (D, Σ) is not in XNF. Without loss of generality, assume that there exists a nontrivial functional dependency $S' \rightarrow p'.@l'$ such that $S' \rightarrow p'.@l' \in (D, \Sigma)^+$ and $S' \rightarrow p' \notin (D, \Sigma)^+$. By Lemma A.6, there is an XML tree T and a path q prefix of p' such that T conforms to D, T satisfies Σ, $tuples_D(T) = \{t_1, t_2\}$, $t_1.S' = t_2.S'$, $t_1.S' \neq \bot$, $t_1.p' \neq t_2.p'$, $t_1.q \neq t_2.q$ and for each $s \in paths(D)$, if q is not a prefix of s, then $t_1.s = t_2.s$. If $t_1.p'.@l' \neq t_2.p'.@l'$, then $(D, \Sigma) \not\vdash S' \rightarrow p'.@l'$, a contradiction. Thus, we can assume that $t_1.p'.@l' = t_2.p'.@l'$. We can also assume $t_1.p'.@l' \neq \bot$, since if $t_1.p'.@l' = t_2.p'.@l' = \bot$, then $t_1.p' = t_2.p' = \bot$ and, therefore, $T \models S' \rightarrow p'$. Define a new tree tuple t_1' as follows: $t_1'.w = t_1.w$, for each $w \neq p'.@l'$, $t_1'.p'.@l' \neq t_1.p'.@l'$ and $t_1'.p'.@l' \neq \bot$. Then, there is an XML tree $T' \in trees_D(\{t_1', t_2\})$ such that $T' \models D$ and $T' \not\models S' \rightarrow p'.@l'$, since $p'.@l' \notin S'$ ($S' \rightarrow p'.@l'$ is a nontrivial functional dependency). If $T' \models \Sigma$, then $(D, \Sigma) \not\vdash S' \rightarrow p'.@l'$, a contradiction. Hence $T' \not\models \Sigma$ and, therefore, there is $S \rightarrow p'' \in \Sigma$ such that $T' \not\models S \rightarrow p''$. But p'' must be equal to $p'.@l'$, since $t_1, t_2 \in tuples_D(T)$ and $T \models \Sigma$. Therefore, $T \not\models S \rightarrow p'$, because $t_1.S = t_1'.S = t_2.S$, $t_1'.S \neq \bot$ and $t_1.p' \neq t_2.p'$. We conclude that $(D, \Sigma) \not\vdash S \rightarrow p'$, which contradicts our initial assumption since $S \rightarrow p'.@l'$ is a nontrivial FD in Σ.