Removing XML Data Redundancies Using Functional and Equality-Generating Dependencies

Junhu Wang[1] **Rodney Topor**[2]

[1]INT, Griffith University, Gold Coast, Australia
J.Wang@griffith.edu.au
[2]CIT, Griffith University, Brisbane, Australia
R.Topor@griffith.edu.au

Abstract

We study the design issues of data-centric XML documents where (1) there are no mixed contents, i.e., each element may have some subelements and attributes, or it may have a *single* value in the form of a character string, but not a mixture of strings and subelements and/or attributes, (2) the ordering of subelements is of no significance. We provide a new definition of functional dependency (FD) for XML that generalizes those published previously. We also define equality-generating dependencies (EGDs) for XML, which, to our knowledge, have not been studied before. We show how to use EGDs and FDs to detect data redundancies in XML, and propose normal forms of DTDs with respect to these constraints. We show that our normal forms are necessary and sufficient to ensure all conforming XML documents have no redundancies. In passing, we define a normal form for relational databases based on EGDs in relational systems that can help remove data redundancies across multiple relations.

Keywords: XML tree, DTD, relation, functional dependency, equality-generating dependency, data redundancy, normal form, normalization.

1 Introduction

It is well known that XML documents can be regarded as a new type of database, and such data are particularly good for information exchange on the internet. The design of XML data has attracted much attention recently. As with any type of database, poorly designed documents may contain too many unnecessary redundancies and these redundancies may cause update anomalies. Data redundancies are usually due to some form of dependencies among the data, such as *functional dependencies (FDs)* and *multi-valued dependencies* in relational databases. Traditional functional dependencies are not suited for XML data because of the structural difference between the two types of database. On the other hand, dependencies naturally exist among data, no matter what format the data is in. Therefore, attempts to define data dependencies for XML and use them in the design of XML database have been made by several groups of researchers. For example, XML functional dependencies (XFDs) have been defined in (Wu, Ling, Y.Lee, Lee & Dobbie 2002, Lee, Ling & Low 2002, Arenas & Libkin 2004, Vincent, Liu & Liu 2004, Hartmann & Link 2003). However, the definitions of XFDs in these works are all different , and they do not always represent the same type of constraints in an XML document. Not surprisingly there is no consensus yet as to which definition is the 'best'. A common problem with these definitions is the limited expressive power. As will be seen later, none of the XFDs in (Wu et al. 2002),(Lee et al. 2002),(Arenas & Libkin 2004), and (Vincent et al. 2004) can express the constraint that the student number determines the set of addresses of the student in the document shown in Figure 1, nor can they express the constraint that the

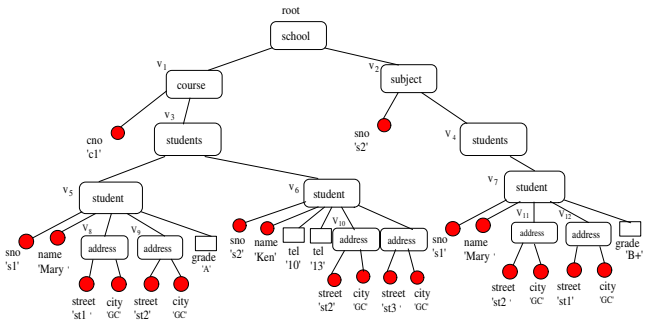

Figure 1: The Course-Subject-Student example

title, authors, and year of publication of a book can determine the publisher in the document shown in Figure 2. The functional dependencies in (Arenas & Libkin 2004) and (Vincent et al. 2004) even cannot express the constraint that the student number determines the student name for *all* student nodes in the *entire document* shown in Figure 1, because the student nodes are located in different paths (under both courses and subjects). The XFDs in (Hartmann

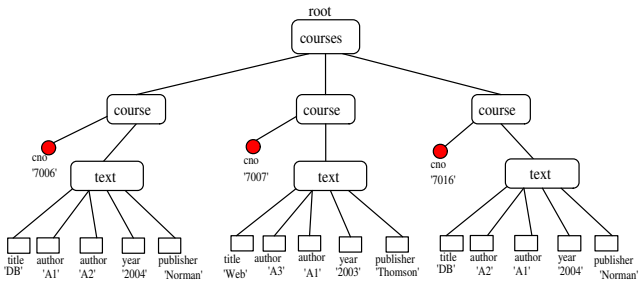

Figure 2: The Course-Text example

& Link 2003), however, cannot express the constraint that the student number (sno) determines the student node in each course, that is, no two different student nodes under the same course can have the same sno. There are also natural constraints that can not be expressed by any of the previously defined XFDs. For example, in Figure 1 if we assume that a student may have several phone numbers (tel), but no phone number is shared by two or more students, then any single

phone number of the student will determine the student and hence the set of addresses of the student. This constraint cannot be expressed by any of the previously defined XFDs mentioned above. For a more detailed survey of related work, see Section 7.

In this paper, we present a new definition of functional dependencies for XML data that can express all of the constraints mentioned in the above examples. Our definition not only captures some useful constraints that can not be expressed before, but also unifies the previous definitions of XFDs under the same framework. We also define *equality-generating dependencies (EGDs)* for XML data, which, to the best of our knowledge, have not been studied before. We show how to use these dependencies (FDs and EGDs) to detect and remove potential data redundancies in XML documents. In passing, we propose a normal form for relational database schemas with respect to EGDs in relational systems that can help remove data redundancies across multiple relations. We believe this is important in itself because traditional normal forms (eg, BCNF, 4NF and 5NF) only eliminate redundancies in a single relation. Our normal form is an direct extension of BCNF.

The rest of the paper is organized as follows. Section 2 provides preliminary definitions and notations. Section 3 presents our new definition of functional dependencies for XML data and shows how our FDs can, for example, express the constraints in the abovementioned examples. Section 4 starts with a discussion of data redundancies across relations in relational databases, and defines a normal form of relational database schema with respect to EGDs that prevents such redundancies. Then it goes on to discuss similar problems in XML data and defines EGDs for XML documents. Section 5 considers FDs and EGDs in the presence of DTDs. Section 6 discusses potential data redundancies in XML documents conforming to a DTD, which are caused by a given set of FDs or EGDs, and proposes normal forms of DTDs that prevent such problems. Section 7 compares our results with related work. We conclude the paper with a discussion about unresolved issues and future work in Section 8.

2 Preliminary Definitions and Notations

2.1 XML tree

In this paper we will consider XML documents where (1) there are no mixed contents, i.e., each element may have some subelements and attributes, or it may have a *single* value in the form of a character string, but not a mixture of strings and subelements and/or attributes, (2) the ordering of subelements is of no significance. It should be mentioned that most data-centric XML documents (eg, those converted from relational and OO databases) have these characteristics. We call elements that only have a single value a *simple* element, and elements that have subelements and/or attributes *complex* elements. Simple elements serve similar purposes to attributes, the only difference is that multiple simple elements with the same label may appear as children of a complex element, but each attribute of a complex element has a distinct label.

As well known, an XML document can be represented by a tree. Figures 1 and 2 are two example XML trees. In the figures, complex elements are shown as rounded rectangles, attributes are shown as filled circles, simple elements are shown as squares, and string values are quoted.

To facilitate our discussion in subsequent sections, we provide a formal definition of XML trees. Let $\mathbf{E_1}$ and $\mathbf{E_2}$ be disjoint sets of element names, $\mathbf{A}$ be a set of attribute names, $\mathbf{E} = \mathbf{E_1} \cup \mathbf{E_2}$, and $\mathbf{E}$ and $\mathbf{A}$ be disjoint. Element names and attribute names are called *labels*.

Definition 2.1 [**XML tree**] An XML tree is defined to be $T = (V, lab, ele, att, val, root)$, where (1) V is a set of nodes; (2) lab is a mapping from V to $\mathbf{E} \cup \mathbf{A}$ which assigns a label to each node in V; a node $v \in V$ is called a *complex element* (node) if $lab(v) \in \mathbf{E_1}$, a *simple element* (node) if $lab(v) \in \mathbf{E_2}$, and an *attribute* (node) if $lab(v) \in \mathbf{A}$. (3) ele and att are functions from the set of complex elements in V: for every $v \in V$, if $lab(v) \in \mathbf{E_1}$ then $ele(v)$ is a set of element nodes, and $att(v)$ is a set of attribute nodes with distinct labels; (4) val is a function that assigns a value to each attribute or simple element. (5) $root$ is the unique root node labelled with complex element name r. (6) If $v' \in ele(v) \cup att(v)$, then we call v' a *child* of v. The parent-child relationships defined by ele and att form a tree rooted at $root$. □

As stated explicitly in the definition, ele and att define the child nodes of a complex element node. Child elements are also referred to as *subelements*, and child attribute nodes are sometimes simply referred to as *attributes*. The concept of *ancestors* and *descendants* are defined as usual: a node v is the ancestor of another node v' if v is the parent of v', or v is the parent of an ancestor of v'; v' is a descendent (node) of v if v is an ancestor of v'.

Note that our definition of XML trees is different from those, for example, in (Buneman, Davidson, Fan, Hara & Tan 2001, Fan, Schwenzer & Wu 2001, Arenas & Libkin 2004). We have explicitly distinguished complex and simple elements so that the special text node under a simple element is not required. We have also made the ordering of child elements insignificant by treating them as a set rather than a sequence.

2.2 Paths in XML trees

We distinguish three types of paths: downward paths, upward paths, and composite paths. These paths are subclasses of XPath.

Definition 2.2 [**paths**] A *downward path* is of the form $l_1.l_2.\cdots.l_n$ where $l_i \in \mathbf{E} \cup \mathbf{A} \cup \{\_, \sim\}$ $(i = 1, \ldots, n-1)$, $l_n \in \mathbf{E} \cup \mathbf{A}$, and if there is an attribute name or a simple element name in the path, it must appear at the last position, that is, it must be l_n. In a downward path the symbol $\_$ represents a wildcard (which can match any label), and $\sim$ represents $\_*$, namely the Kleene closure of the wildcard. A *simple path* is a downward path where there is no $\_$ or $\sim$. The number of labels in a simple path p is called its *length*. The simple path of length 0 is called the *empty path* and denoted ϵ.

An *upward path* is of the form $\Uparrow.\cdots.\Uparrow$. If there are $k > 1$ upward arrows (to distinguish from the empty path, we require $k \geq 1$), we will sometimes abbreviate the path as $\Uparrow^k$.

A *composite path* is of the form $\xi.\rho$, where ξ is an upward path, and ρ is a simple path. □

According to the above definition, an upward path is a special composite path, the empty path is a special simple path, which in turn is a special downward path.

A *path* is either a downward path or a composite path. Let us use $last(p)$ to denote the last symbol in path p.

In any XML tree T, only some paths are valid. Here the validity of paths is with respect to a node, as defined below.

Definition 2.3 [**valid paths**]Let T be an XML tree and v_0 be a node in T. A path p is said to be *valid wrt* v_0, if one of the following is true:

- p is the empty path.
- p is the downward path $l_1.\cdots.l_n$, and there is a sequence of nodes $v_1,\ldots,v_n$ in T such that for $i=1,\ldots,n$ (1) if l_i is in $\mathbf{E}\cup\mathbf{A}\cup\{\_\}$, then v_i is a child of v_{i-1} and the label of v_i matches l_i (note that the label of any node matches _); (2) if l_i is $\sim$, then v_i is v_{i-1} or a descendant of v_{i-1}.
- p is the composite path $\Uparrow^k .l_1.\cdots.l_n$ and there is a sequence of nodes $n_1,\ldots,n_k,v_1,\ldots,v_n$ in T such that n_1 is the parent of v_0, n_i is the parent of n_{i-1} for $i=2,\ldots,k$, and v_1 is a child of n_k, v_i is a child of v_{i-1} for $i=2,\ldots,n$, and the label of v_i matches l_i for $i=1,\ldots,n$.

□

The sequence of nodes described above (for a non-empty path p) is called an *instance of path p wrt v_0*. Generally there may be many such instances. We will refer to the set

$$\{v_n \mid v_n \text{ is the last node of an instance of } p \text{ wrt } v_0\}$$

as the *target set of p wrt v_0*, denoted $v_0[p]$. For easy presentation, we also define $v_0[\epsilon]=\{v_0\}$ for any node v_0. Note that path p is not valid wrt v_0 if and only if $v_0[p]$ is empty.

For example, in the XML tree shown in Figure 1, student.sno and student.address.city are simple paths that are valid wrt v_3 and v_4, but not to others; $\Uparrow.\Uparrow$ is an upward path valid wrt all nodes except v_1, v_2 and *root*; $\Uparrow$.cno is a composite path which is valid wrt v_3; and *root*[course.students.student] $=\{v_5,v_6\}$, *root*[$\sim$.student] $=\{v_5,v_6,v_7\}$, v_5[address] $=\{v_8,v_9\}$, and v_5[address.street] is the set containing the two street attribute nodes under v_8 and v_9.

2.3 Value Equality and Node Agreement

In order to compare two nodes n_1 and n_2 in an XML tree, we need to define the equality between them. Obviously, if n_1 and n_2 are the same node (denoted $n_1=n_2$), they should be considered equal, but this kind of *node equality* is not sufficient, because there are cases where two *distinct* nodes have equal *values*. So we need to define *value equality* between nodes. Since we consider the ordering of child elements insignificant, our definition of value equality is different from that in (Buneman, Davidson, Fan & Hara 2002, Buneman et al. 2001).

Definition 2.4 [**value equal**] Let n_1 and n_2 be two nodes in T. We say n_1 and n_2 are *value equal*, denoted $n_1=_v n_2$, if n_1 and n_2 are of the same label, and

1. n_1 and n_2 are both attribute nodes or simple element nodes, and the two nodes have the same value, or
2. n_1 and n_2 are both complex elements, and for every child node m_1 of n_1, there is a child node m_2 of n_2 such that $m_1=_v m_2$, and vice versa.

□

For example, the two nodes v_8 and v_{12} in Figure 1 are value equal. The two nodes v_9 and v_{11} are also value equal.

Note that node equality implies value equality, but not vice versa.

We now turn to the definition of the *agreement* of two nodes on a path. Intuitively, given two nodes n_1 and n_2 and a path p, there can be several different interpretations of agreements between n_1 and n_2 on p. For example, every node in the target set $n_1[p]$ may have a node in $n_2[p]$ such that the two nodes are value equal and vice versa, or there may be only some nodes in the two sets that are value equal. In this paper, we are interested in the cases defined below.

Definition 2.5 [**types of agreement**] Let n_1,n_2 be two nodes with the same label. Let p be a simple or composite path.

- We say n_1 and n_2 *node agree* or *N-agree* on p if
  - p is a simple path, and $n_1=n_2$; or
  - p is an upward path, $n_1[p]\neq\emptyset$ and $n_1[p]=n_2[p]$; or
  - p is a composite path, and n_1 and n_2 node agree on the upward path part of p.
- We say that n_1 and n_2 *set agree* or *S-agree* on p if for every node v_1 in $n_1[p]$, there is a node v_2 in $n_2[p]$ such that $v_1=_v v_2$, and vice versa.
- We say that n_1 and n_2 *intersect agree* or *I-agree* on p if there exist nodes $v_1\in n_1[p]$ and $v_2\in n_2[p]$ such that $v_1=_v v_2$.

□

For example, in Figure 1, v_5 and v_7 S-agree on address; v_6 and v_7 I-agree on address; v_5,v_6 N-agree on $\Uparrow$.

It is straightforward to see that N-agreement implies S-agreement, which in turn implies I-agreement. Besides, using induction on the length of paths, we can easily prove the following lemmas.

Lemma 2.1 *Let n_1 and n_2 be two nodes. Then $n_1=_v n_2$ if and only if n_1 and n_2 S-agree on every simple path.*

Lemma 2.2 *Let n_1 and n_2 be two nodes, and $p_1\equiv l_1.\cdots.l_m$ and $p_2\equiv l_1.\cdots.l_m.l_{m+1}$ be two simple or composite paths, where l_{m+1} is not $\Uparrow$. Then*

- *n_1 and n_2 N-agree on p_2 implies they N-agree on p_1.*
- *n_1 and n_2 S-agree on p_1 implies they S-agree on p_2.*
- *n_1 and n_2 I-agree on p_1 implies they I-agree on p_2.*

3 Functional Dependencies

Our definition of functional dependency uses the different types of agreements on paths that are defined in the previous section.

Definition 3.1 [**XML functional dependency**] Let T be an XML tree. A functional dependency (FD) on T is an expression of the form

$$Q:\ p_1(c_1),\ldots,p_n(c_n)\ \rightarrow p_{n+1}(c_{n+1})$$

where Q is a downward path, $p_1,p_2,\ldots,p_n$ are simple or composite paths, p_{n+1} is a simple path of length 1 or 0, and c_i $(i=1,\ldots,n+1)$ is one of N, S, and I.

T is said to *satisfy* the above functional dependency if, for any two nodes $n_1,n_2\in$ *root*$[Q]$ the following statement is true: if $n_1[p_i]$ is not empty, and n_1,n_2 c_i-agree on p_i (for all $i=1,\ldots,n$), then $n_1[p_{n+1}]$ and $n_2[p_{n+1}]$ are not empty, and n_1 and n_2 also c_{n+1}-agree on p_{n+1}. □

In Definition 3.1 we require that the type of agreement be specified for every path. However, some types of agreements are more common than others in practical cases. To simplify the notation, when c_i is omitted we use the following default types of agreement: for the empty path or an upward path, the default is N-agreement; for all other paths, the default is S-agreement.

Our definition of XFD generalizes those in previously published work, and it can express many different constraints, some of which cannot be expressed by any of the previous XFDs. Here we provide a few examples only.

Example 3.1 We use the XML tree in Figure 1 in this example.

(1) To say that any single telephone number of a student determines his/her set of addresses, we can use

$\sim .student : tel(I) \rightarrow address(S)$.

This constraint cannot be expressed by any of the previously defined XFDs.

(2) To say that student number determines student name for all student nodes, we can use

$\sim .student : sno \rightarrow name$.

This constraint can not be expressed using the functional dependencies defined in (Arenas & Libkin 2004) or (Vincent et al. 2004) because the student nodes are located in different paths.

(3) To say that the student number determines the student's set of addresses, we can use

$\sim .student : sno \rightarrow address$.

Note that this is different from the multi-valued dependencies defined in (Vincent & Liu 2003), and it cannot be expressed by the XFDs in (Arenas & Libkin 2004), (Vincent et al. 2004), (Lee et al. 2002), or (Wu et al. 2002).

(4) To say that the course number determines the course node, i.e., no two course nodes have the same course number, we can use

$course : cno \rightarrow \epsilon$.

This constraint cannot be expressed by the XFDs in (Hartmann & Link 2003).

(5) To say that the course code and student number determines the student grade in that course, we can use

$course.students.student : sno, \Uparrow^2 .cno \rightarrow grade$.

□

Example 3.2 Let us now look at the XML tree about textbooks shown in Figure 2. Suppose that the title, the set of authors, and the year of publication can determine the publisher. This constraint can be expressed by the functional dependency

$course.text : title, author, year \rightarrow publisher$.

The above constraint can not be expressed using the FDs defined in any of the previous work mentioned above except (Hartmann & Link 2003). □

Note that, according to definition, if $root[Q] = \emptyset$, that is, Q is not valid wrt $root$, then the FD is trivially satisfied by T. Also when checking satisfaction of a FD, we only need to consider those nodes in $root[Q]$ that have a non-empty target set for every path on the LHS. In particular, if there is a path on the LHS which is not valid to any node in $root[Q]$, then the functional dependency is trivially satisfied by T. Although it appears to be useless to consider invalid paths, this is actually necessary when we define FD assertions on DTDs later.

4 Equality-Generating Dependencies

We start with a discussion of EGDs in relational databases first, and define a normal form that extends BCNF to multiple relations. Then we point out similar problems in XML and define EGDs for XML data.

4.1 EGDs in Relational Databases

In relational databases, the traditional normalization technique removes data redundancies within a single relation, but it cannot remove redundancies across relations. For example, if we have two relations

$Graduate(sNo, sName, address)$

$UMember(stNum, stName, phone)$

representing graduate students and student union members, where *sNo* and *stNum* both represent student number, *sName* and *stName* both represent student name, then although both relations are in BCNF, there are data redundancies across the two relations if the student information in the two relations overlap. Such redundancies can be detected using *equality-generating dependencies (EGDs)*. An EGD[1] is an expression of the form

$$R_1.X_1 = R_2.X_2 \rightarrow R_1.Y_1 = R_2.Y_2 \qquad (*)$$

where R_1, R_2 are two relation schemas, X_1, Y_1 are *lists*[2] of attributes in R_1, and X_2, Y_2 are lists of attributes in R_2. The EGD specifies that, for any two tuples t_1 and t_2 in instances of R_1 and R_2 respectively, whenever $t_1[X_1] = t_2[X_2]$, then $t_1[Y_1] = t_2[Y_2]$. For instance, for the example above, there is an EGD

$$Graduate.sNo = UMember.stNum \rightarrow$$
$$Graduate.sName = UMember.stName$$

Due to the above EGD, if the two relations overlap, then there will be data redundancy. To remove such redundancies we can restructure the two tables as follows: if every union member appears in the graduate table, we can change the *UMember* table to $UMember(stNum, phone)$ and add a foreign key "$UMember(stNum)$ references $Graduate(sNo)$"; if only some graduates are union members, and only some union members are graduates, we can add another relation $Student(sNo, sName)$, modify the original tables to $Graduate(sNo, address)$ and $UMemeber(sNo, phone)$, and add the foreign keys "$Graduate(sNo)$ references $Student(sNo)$" and "$UMemeber(sNo)$ references $Student(sNo)$".

FDs are special EGDs where $R_1 = R_2, X_1 = X_2$, and $Y_1 = Y_2$. Like FDs, an EGD can be trivial or non-trivial. An EGD is said to be *trivial* if it holds in every database schema. For example, every trivial FD is a trivial EGD. Also, the EGD (*) will be trivial if Y_1 is a sublist of X_1, and Y_2 is a corresponding sublist of X_2.

Given a set E of EGDs for a database schema $\mathbb{D}$, we may be able to derive some other EGDs. For example, from $R_1.x = R_2.x \rightarrow R_1.y = R_2.y$ and $R_1.y = R_2.y \rightarrow R_1.z = R_2.z$ we can derive $R_1.x = R_2.x \rightarrow R_1.z = R_2.z$. Let us use $(\mathbb{D}, E)^+$ to denote the set of all EGDs that hold in $\mathbb{D}$ and that can be derived from E.

We now define a new normal form of a relational database schema that directly extends BCNF.

[1] For simplicity, we consider only EGDs involving two tuples on the left-hand side, but the idea presented here readily extends to general EGDs. Also note our EGDs are not defined on a universal relation as in (Fagin & Vardi 1984).

[2] We use list to stress that (1) the attributes in X_1 (and Y_1) are ordered, and (2) an attribute in X_1 (and Y_1) may appear more than once.

Definition 4.1 [**normal form wrt EGDs in relational databases**] A relational database schema $\mathbb{D}$ is said to be in *normal form* with respect to a given set E of EGDs, if for every non-trivial EGD

$$R_1.X_1 = R_2.X_2 \rightarrow R_1.Y_1 = R_2.Y_2$$

in $(\mathbb{D}, E)^+$,

- if $R_1 = R_2, X_1 = X_2$, and $Y_1 = Y_2$ then X_1 is a superkey of R_1;
- Otherwise, there is a corresponding *exclusion constraint* $R_1[X_1] \cap R_2[X_2] = \emptyset$, which means that the projections $r_1[X_1]$ and $r_2[X_2]$ are disjoint, where r_1 and r_2 are instances of R_1 and R_2 respectively in every possible database instance.

□

We now briefly discuss about the above definition. For the EGD in the definition, if $R_1 = R_2, X_1 = X_2$ and $Y_1 = Y_2$, it becomes a FD $R_1 : X_1 \rightarrow Y_1$. By requiring X_1 to be a superkey of R_1 we are demanding that R_1 be in BCNF with respect to the FD. Therefore, the first condition in our normal form is equivalent to say that all relation schema is in BCNF with respect to the FDs in $(\mathbb{D}, E)^+$. If $R_1 \neq R_2$, or $X_1 \neq X_2$, or $Y_1 \neq Y_2$, then the EGD is not a FD, and the second condition in our normal form requires that instances of R_1 and R_2 must not overlap on the X_1 (X_2) attributes. In effect, in both cases we require that there are no distinct tuples $t_1 \in r_1$ and $t_2 \in r_2$, where r_1 and r_2 are instances of R_1 and R_2 respectively, such that $t_1[X_1] = t_2[X_2]$. In addition, if $R_1 = R_2$, but $X_1 \neq X_2$, we also require that $t[X_1] \neq t[X_2]$ for every tuple t. In all cases, the normal form requires there is a constraint on the database schema which makes sure that the pre-condition (i.e, the equality on the left hand side) of every EGD can not be satisfied by any non-empty database instance.

In our graduate student/union member example above, the database schema is in normal form with respect to the given EGD if and only if $Graduate[sNo] \cap UMemembr[stNum] = \emptyset$ holds. That is, there are no graduate students who is a union member.

For a detailed discussion about the inference rules for relational EGDs and a lossless decomposition algorithm that decomposes a relational schema into one in normal form with respect to a set of EGDs, see (Wang 2004).

4.2 EGDs for XML

EGDs may also exist and cause redundancies in XML data. Figure 3 shows an example document where a student union member also has student number, name and telephone, which are already stored under some student node.

Before defining EGDs for XML, we need to revise our definition of value equality to include nodes with *literally different* but *semantically identical* labels. Ideally, we should have a complete classification of all labels such that labels with the same meaning are put to the same set, and those with different meanings (even if they are literally identical, eg, name of a product and name of a supplier) are put to different sets. There are many possible ways to make such a classification, ranging from trivially dividing the labels into disjoint sets to sophisticated classifications using *Ontology* (which, for instance, also checks the position of occurrences of the labels) (Sowa n.d.). Here we just assume such a classification exists, and we use $l_1 \doteq l_2$ to denote that label l_1 and l_2 are semantically identical.

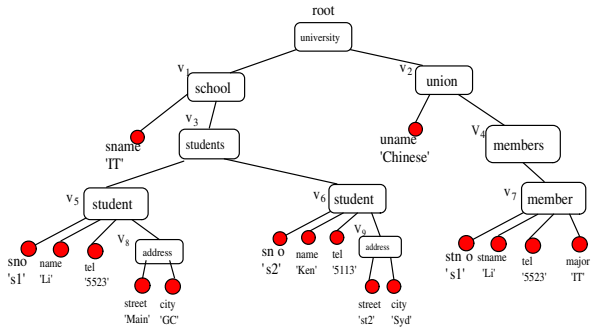

Figure 3: The Uni-School-Union example

Definition 4.2 [**semantic value equal**] Let n_1 and n_2 be two nodes in XML tree T. We say n_1 and n_2 are *semantically value equal*, denoted $n_1 =_{sv} n_2$, if $lab(n_1) \doteq lab(n_2)$, and

1. n_1 and n_2 are both attribute nodes or simple elements, and the two nodes have the same value, or
2. n_1 and n_2 are both complex elements, and for every child node a_1 of n_1, there is a child node a_2 of n_2 such that $a_1 =_{sv} a_2$, and vice versa.

□

For example, in the XML tree in Figure 3, assuming $sno \doteq stno$, and $name \doteq stname$, then the *sno* attribute node under v_5 and the *stno* attribute node under v_7 are semantically value equal, and so are the two attributes *name* and *stname* under v_5 and v_7 respectively.

Let S_1 be the paths $p_1, \ldots, p_n$ and S_2 be the paths $q_1, \ldots, q_n$. Let n_1 and n_2 be two nodes. We will use $n_1.S_1 =_{sv} n_2.S_2$ to denote the fact that $n_1[p_i] \neq \emptyset$, $n_2[p_i] \neq \emptyset$, and every node v in $n_1[p_i]$ has a corresponding node v' in $n_2[q_i]$ such that $v' =_{sv} v$ and vice versa, for all $i = 1, \ldots, n$.

Definition 4.3 [**XML equality-generating dependency**] Let T be an XML Tree. An equality-generating dependency (EGD) on T is an expression of the form

$$Q_1, Q_2 : \ 1.S_1 =_{sv} 2.S_2 \rightarrow 1.q_1 =_{sv} 2.q_2$$

where Q_1 and Q_2 are downward paths, S_1, S_2 are lists of simple or composite paths, and q_1, q_2 are simple paths of length 1 or 0.

T is said to satisfy the EGD if for every pair of nodes $n_1 \in root[Q_1]$ and $n_2 \in root[Q_2]$ the following statement is true: if $n_1[p] \neq \emptyset$ for every $p \in S_1$, and $n_1.S_1 =_{sv} n_2.S_2$, then $n_1[q_1] \neq \emptyset$, $n_2[q_2] \neq \emptyset$, and $n_1.q_1 =_{sv} n_2.q_2$. □

Example 4.1 The XML tree in Figure 3 satisfies the following EGDs:

$school.students.student, union.members.member$:
$1.sno =_{sv} 2.stno \rightarrow 1.name =_{sv} 2.stname$,

$school.students.student, union.members.member$:
$1.sno =_{sv} 2.stno \rightarrow 1.tel =_{sv} 2.tel$ □

A functional dependency where the type of agreement is limited to S-agree can be regarded as a special case of EGD with $Q_1 = Q_2$, $S_1 = S_2$ and $q_1 = q_2$, assuming that for all nodes n_1, n_2, $lab(n_1) \doteq lab(n_2)$ iff $lab(n_1) = lab(n_2)$.

5 Dependencies over DTDs

DTDs and XML Schema documents (we call them XML *scheme* files) can be used to restrict the structure of XML documents. These files define the legal paths, among other things, in a conforming XML document. On top of a scheme file, we can put further restrictions on the data in conforming documents by insisting that some FDs or EGDs must hold. We call these dependencies *assertions*.

For simplicity we will focus on DTDs in this paper, but the ideas presented here also apply to any scheme file including XML Schema documents. The following definition of DTDs is *adapted* from (Arenas & Libkin 2004).

Definition 5.1 [**DTD and natural paths**] A DTD is defined to be $\mathcal{D} = (E_1, E_2, A, P, R, r)$ where $E_1 \subseteq \mathbf{E_1}$ is a finite set of complex element names; $E_2 \subseteq \mathbf{E_2}$ is a finite set of simple element names; $A \subseteq \mathbf{A}$ is a finite set of attributes; P is a mapping from E_1 to element type definitions: $\forall \tau \in E_1$, $P(\tau)$ is a regular expression

$$\alpha = \varepsilon \mid \tau' \mid \alpha|\alpha \mid \alpha, \alpha \mid \alpha^*$$

where ε is the empty sequence, $\tau' \in E_1 \cup E_2$, and "|" , ",", and "*" denote union, concatenation, and the Kleene closure; R is a mapping from E_1 to sets of attributes; r is the element type of the root, which is distinct from all other symbols.

A *natural path* in $\mathcal{D}$ is a string $l_1.\cdots.l_m$, where l_1 is in the alphabet of $P(r)$, l_i is in the alphabet of $P(l_{i-1})$ for $i \in [2, m-1]$, l_m is in the alphabet of $P(l_{m-1})$ or in $R(l_{m-1})$. The set of all natural paths in $\mathcal{D}$ is denoted $paths(\mathcal{D})$. □

The conformity of an XML tree to a DTD is defined as follows.

Definition 5.2 An XML tree $T = (V, lab, ele, att, val, root)$ is said to *conform* to an XML scheme file $\mathcal{S} = (E_1, E_2, A, P, R, r)$ if

1. $lab(root) = r$,
2. lab maps every node in V to $E_1 \cup E_2 \cup A$.
3. for every complex element node $v \in V$, if $ele(v) = \{v_1, \ldots, v_k\}$, then a permutation of the sequence $lab(v_1), \ldots, lab(v_k)$ must be in the language defined by $P(lab(v))$; if $att(v) = \{v'_1, \ldots, v'_m\}$ then $lab(v'_1), \ldots, lab(v'_m)$ must be in the set $R(lab(v))$.

□

Clearly if XML tree T conforms to DTD $\mathcal{D}$, then every simple path of T, if valid wrt the root, is in $paths(\mathcal{D})$.

Note that in an abstract DTD of Definition 5.1, there are no constraints such as `ID` or `IDREFS` that may exist in a DTD written according to the W3C specification. Mixed contents are not allowed either. On the other hand, every abstract DTD has an equivalent W3C DTD. Since the W3C DTDs are more familiar to readers, we will use them in our examples.

Figure 4 shows an example DTD. The XML tree in Figure 1 conforms to the DTD.

As mentioned earlier, we can make assertions over a DTD $\mathcal{D}$. Informally, an assertion is an FD or EGD defined on conforming XML trees, and it asserts that every conforming XML tree T must satisfy the dependency. For an FD or EGD to qualify as an assertion, the paths it contains must be *legal paths* in $\mathcal{D}$, as defined below.

```
<!ELEMENT school (course|subject)* >
<!ELEMENT course (students+) >
<!ATTLIST course
        cno CDATA #REQUIRED>
<!ELEMENT subject (students+) >
<!ATTLIST subject
        sno CDATA #REQUIRED>
<!ELEMENT students (student)*>
<!ELEMENT student (tel*, address+, grade?)>
<!ATTLIST student
        sno CDATA #REQUIRED
        name CDATA #REQUIRED>
<!ELEMENT tel (#PCDATA)>
<!ELEMENT address EMPTY>
<!ATTLIST address
        street CDATA #REQUIRED
        city  CDATA #REQUIRED>
<!ELEMENT grade  (#PCDATA)>
```

Figure 4: The Course-Subject-Student DTD

Definition 5.3 [**legal path**] The *legal paths* in a DTD are defined as follows:

- A substring (including the empty substring) of a natural path is a legal path, called a *legal simple path*;
- if $p = l_1.\cdots.l_m$ is a natural path, then the string obtained by replacing some l_i with _ or by replacing some substring of p with $\sim$ is a legal path, called a *legal downward path*;
- if there is a legal simple path of length $k > 0$, then $\Uparrow^k$ is a legal path, called a *legal upward path*.
- if p is a natural path (which may have a length of 0 or more), and $p.l_1.\cdots.l_j$ and $p.l'_1.\cdots.l'_k$ are natural paths, then
  - $\Uparrow^j .l'_1.\cdots.l'_k$ is a legal path, called a *legal composite path*;
  - if $p'.l_j$ is a legal downward path obtained from the natural path $p.l_1.\cdots.l_j$, then $p'.l_j.\ \Uparrow^j$ and $p'.l_j.\ \Uparrow^j .l'_1.\cdots.l'_k$ are legal paths.

□

Intuitively, a path is a legal path if it is valid wrt some node in at least one conforming document. For example, in the DTD of Figure 4, all of the following are legal paths. $\sim$ *.student*, $\sim$ *.student.sno*, $\sim$ *.student.* $\Uparrow^2$ *.cno*. But *name.cno* is not a legal path.

We can now formally define assertions.

Definition 5.4 [**Assertion**] Given a DTD $\mathcal{D}$, an *assertion* $\mathcal{A}$ over $\mathcal{D}$ is either of the following:

(1) a FD assertion

$$Q:\ p_1(c_1), \ldots, p_n(c_n)\ \rightarrow p_{n+1}(c_{n+1})$$

where Q is a legal downward path, $p_1, \ldots, p_n$ are legal simple, upward or composite paths, p_{n+1} is a legal simple path of length 1 or 0, $Q.p_i$ is a legal path, and $c_i \in \{N, S, I\}$, for $i \in [1, n+1]$.

(2) an EGD assertion

$$Q_1, Q_2:\ 1.S_1 =_{sv} 2.S_2 \rightarrow 1.q_1 =_{sv} 2.q_2$$

where Q_1 and Q_2 are legal downward paths, q_1, q_2 are legal simple paths of length 0 or 1, S_1 and S_2 are lists of legal simple, upward, or composite paths, and for $i = 1, 2$ and $p_j \in S_i \cup \{q_i\}$, $Q_i.p_j$ is a legal path.

The assertion states that every XML tree T conforming to $\mathcal{D}$ must satisfy the corresponding dependency. □

It is easy to see the FDs in Example 3.1 are assertions over the DTD in Figure 4.

Figures 5 and 6 show two more DTDs. The XML trees in Figures 2 and 3 conform to the two DTDs respectively. The FD in Example 3.2 is an assertion over the DTD in Figure 5, and the EGDs in Example 4.1 are assertions over the DTD in Figure 6.

```
<!ELEMENT courses (course+)>
<!ELEMENT course (text)>
   <ATTLIST course
     cno #REQUIRED >
<!ELEMENT text (title, author+, year, publisher)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
```

Figure 5: The Course-Text DTD

```
<!ELEMENT university (school+, union*)>
<!ELEMENT school (students)>
<!ATTLIST school
          sname #ID>
<!ELEMENT students (student+)>
<!ELEMENT student (address)>
<!ATTLIST student
          sno CDATA #REQUIRED
          name CDATA #REQUIRED
          tel CDATA #IMPLIED>
<!ELEMENT address EMPTY>
<!ATTLIST address
          street CDATA #REQUIRED
          city  CDATA #REQUIRED>
<!ELEMENT members (member+)>
<!ELEMENT member EMPTY>
<!ATTLIST member
          stno CDATA #REQUIRED
          stname  CDATA #REQUIRED
          tel CDATA #IMPLIED
          major  CDATA #IMPLIED>
```

Figure 6: The Uni-School-Union DTD

An assertion may or may not add any restrictions to the data in conforming XML documents. For example, the assertion $\sim .student : sno \rightarrow sno$ does not add anything to the DTD in Figure 4 because every instance conforming to the DTD will automatically satisfy the assertion. We refer to those assertions that do not add restrictions on the data as trivial assertions. Formally, we have

Definition 5.5 An assertion $\mathcal{A}$ over DTD $\mathcal{D}$ is said to be trivial if every XML tree conforming to $\mathcal{D}$ automatically satisfies $\mathcal{A}$. □

For example, the assertions mentioned above on the three DTDs are all non-trivial. On the other hand, using Lemma 2.1 and 2.2 we can show that the following FD assertions are all trivial (assuming $Q.l_1.\cdots.l_n.l_{n+1}$ is a natural path):

- $Q : l_1.\cdots.l_n(c) \rightarrow l_1.\cdots.l_n(c)$
- $Q : l_1.\cdots.l_n(N) \rightarrow l_1.\cdots.l_n(S)$
- $Q : l_1.\cdots.l_n(S) \rightarrow l_1.\cdots.l_n(I)$
- $Q : l_1.\cdots.l_n.l_{n+1}(N) \rightarrow l_1.\cdots.l_n(N)$
- $Q : l_1.\cdots.l_n(S) \rightarrow l_1.\cdots.l_n.l_{n+1}(S)$
- $Q : l_1.\cdots.l_n(I) \rightarrow l_1.\cdots.l_n.l_{n+1}(I)$

Given a set $\mathcal{F}$ of assertions on DTD $\mathcal{D}$, we may infer other dependencies that must be satisfied by all conforming documents of $\mathcal{D}$. As usual, the set of all dependencies that can be derived from $\mathcal{D}$ and $\mathcal{F}$ is denoted $(\mathcal{D}, \mathcal{F})^+$.

6 XML Normal Forms and Data Redundancy

6.1 Normal Form with respect to FD Assertions

In relational databases, a table is in BCNF with respect to a set of functional dependencies if the left hand side (LHS) of every non-trivial functional dependency is a superkey. In XML documents, we can define a normal form along the same line.

We first define *keys* in XML. Like in relational databases, a key is a special FD.

Definition 6.1 [**key**] If there is a FD assertion $Q : p_1(c_1), \cdots, p_n(c_n) \rightarrow \epsilon$ over the DTD $\mathcal{D}$, then we call $p_1(c_1), \cdots, p_n(c_n)$ a *key* (of Q). □

Intuitively, $p_1(c_1), \cdots, p_n(c_n)$ is a key of Q means that, for every XML tree T conforming to $\mathcal{D}$, and for any two nodes n_1 and n_2 in $r[Q]$ (of T), if n_1 and n_2 c_i-agree on p_i for all $i \in [1, n]$, then n_1 and n_2 must be the same node.

Our definition of keys differs from those in (Buneman et al. 2002), (Buneman et al. 2001) and (Fan et al. 2001) in two ways. First, we did not consider the ordering of subelements to be of significance, and hence the definitions of *value equality* are different. Second, we allow several different interpretations of *agreements* (as represented by the c_is in the definition), while the previous definitions only consider I-agreement (i.e., every c_i is I).

Definition 6.2 [**XML normal form wrt FDs**] A DTD $\mathcal{D}$ is said to be in normal form with respect to a set of FD assertions $\mathcal{F}$, if for every no-trivial assertion in $(\mathcal{D}, \mathcal{F})^+$, the LHS is a key. □

For example, the DTD in Figure 4 is not in normal form with respect to the FD assertion

$\sim .student : sno \rightarrow name$,

because the assertion is non-trivial and we cannot derive $\sim .student : sno \rightarrow \epsilon$ from the DTD and the given assertion. Similarly, the DTD in Figure 5 is not in normal form with respect to the FD assertion

$course.text : title, author, year \rightarrow publisher$.

An alternative definition of normal form is as in Definition 6.3. Recall that nodes n_1 and n_2 N-agree on any simple path p_{n+1} is defined to mean $n_1 = n_2$. Therefore Definition 6.3 is equivalent to Definition 6.2.

Definition 6.3 [**XML normal form wrt FDs**] A DTD $\mathcal{D}$ is said to be in normal form with respect to a set of FD assertions $\mathcal{F}$, if for every no-trivial assertion

$$Q : p_1(c_1), \cdots, p_n(c_n) \rightarrow p_{n+1}(c_{n+1})$$

in $(\mathcal{D}, \mathcal{F})^+$, the functional dependency

$$Q : p_1(c_1), \cdots, p_n(c_n) \rightarrow p_{n+1}(N)$$

is also in $(\mathcal{D}, \mathcal{F})^+$. □

6.2 Normal Forms with Respect to EGD Assertions

Similar to the normal form with respect to EGDs in relational databases defined in Section 4.1, we can define a normal form with respect to EGDs for DTDs.

Definition 6.4 [**XML normal form wrt EGDs**] A DTD $\mathcal{D}$ is said to be in normal form with respect to a set $\mathcal{F}$ of EGD assertions if for every non-trivial EGD

$$Q_1, Q_2 : \ 1.S_1 =_{sv} 2.S_2 \rightarrow 1.q_1 =_{sv} 2.q_2$$

in $(\mathcal{D}, \mathcal{F})^+$,

(1) if $Q_1 = Q_2, S_1 = S_2,$ and $q_1 = q_2$, then S_1 is a key of Q_1.

(2) otherwise the following *disjoint constraint* holds:

$$Q_1, Q_2 : \ 1.S_1 =_{sv} 2.S_2 \rightarrow False$$

which means that in every conforming XML tree, there can not be two nodes $n_1 \in r[Q_1]$ and $n_2 \in r[Q_2]$ such that $n_1.S_1 =_{sv} n_2.S_2$.

□

For example, the DTD in Figure 6 will be in normal form with respect to the EGD assertion

$school.students.student, union.members.member$:
$1.sno =_{sv} 2.stno \rightarrow 1.name =_{sv} 2.stname$

if and only if the following constraint hold:

$school.students.student, union.members.member$:
$1.sno =_{sv} 2.stno \rightarrow 1.\epsilon = 2.\epsilon$

which means that there are no student node s and member node m such that $s.sno =_{sv} m.stno$. In other words, school students and union members must not overlap.

6.3 Data Redundancies

Normal forms are aimed to reduce data redundancies caused by the assertions. In this section we provide a result on the relationship between our DTD normal forms and XML data redundancies.

We need to formally define data redundancy in XML trees first.

Definition 6.5 [**Data Redundancy in XML**] Let $\mathcal{D}$ be a DTD, $\mathcal{F}$ be a set of assertions over $\mathcal{D}$, and T be an XML tree conforming to $(\mathcal{D}, \mathcal{F})$ (i.e., T conforms to $\mathcal{D}$ and satisfies $\mathcal{F}$). We say that T has data redundancies with respect to $\mathcal{F}$ if there is a node n of T such that the subtree rooted at n , if removed from T, can be fully recovered using other parts of T, $\mathcal{D}$, and the assertions in $\mathcal{F}$. That is, we can construct a tree T_1 (to be rooted at the position of n) such that n and the root of T_1 are value equal. □

For example, the XML tree in Figure 2, which conforms to the DTD in Figure 5 and satisfies the assertion in Example 3.2, has data redundancy. This is because we can restore the publisher node under the rightmost text node if it is removed, by using the DTD and the assertion as well as the publisher node under the leftmost text node. Similarly, the XML tree in Figure 1 which conforms to the DTD in Figure 4 and the FD assertion $\sim .student : \ sno \rightarrow address$ has data redundancies. The XML tree in Figure 3 which conforms to the DTD in Figure 6 and the EGD assertion in Example 4.1 has data redundancies, because if we remove the stname attribute under node v_7, we can restore it from the corresponding attribute under v_5.

The next theorem explains why normal-form DTDs are preferred.

Theorem 6.1 *Let $\mathcal{D}$ be a DTD, $\mathcal{F}$ be a set of FD assertions over $\mathcal{D}$, and $\mathcal{E}$ be a set of EGD assertions over $\mathcal{D}$. Then*

- *There exists an XML tree conforming to $(\mathcal{D}, \mathcal{F})$ which has data redundancies iff $\mathcal{D}$ is not in normal form with respect to $\mathcal{F}$.*
- *There exists an XML tree conforming to $(\mathcal{D}, \mathcal{E})$ which has data redundancies iff $\mathcal{D}$ is not in normal form with respect to $\mathcal{E}$.*

It is important to note that we cannot claim that every XML tree conforming to $(\mathcal{D}, \mathcal{F})^+$ will have data redundancy if $\mathcal{D}$ is not in normal form. This is particularly true if some FDs use I-agreement on the RHS.

7 Comparison with Related Work

Apparently the earliest work on using XML functional dependencies in the normalization of XML documents appeared in (Wu et al. 2002). The paper defines an XML normal form based on *partial* and *transitive* dependencies that try to resemble those in the relational system. The authors made the assumption that the scheme tree of the XML document has the property that there is a unique path from the root to every label. Unfortunately this assumption is not very realistic because, for example, both a subject and a student may have an attribute "sno" as in Figure 4. Clearly the two attributes have different paths from the root. Furthermore, the FDs are limited in expressive power because of the limitation the authors put on the types of agreements of two nodes. In effect, only I-agreement is allowed. More recently, (Lee et al. 2002), (Hartmann & Link 2003), (Arenas & Libkin 2004), and (Vincent et al. 2004), and (the later two both have earlier conference versions) all provide definitions of XML functional dependencies, and (Arenas & Libkin 2004) and (Vincent et al. 2004) went further to define XML normal forms based on their FDs.

(Lee et al. 2002) uses XPaths to define functional dependencies among a set of XML subtrees. An XFD is defined as an expression $(Q, [e_1, \cdots, e_n \rightarrow e_{n+1}])$ where Q is an XPath, and e_i, for $i \in [1, n+1]$, is either an element or an element followed by dot and a set of *key attributes* of the element. An XML tree is said to satisfy the XFD if for any two subtrees rooted at a node in $root[Q]$, if they agree on the value of $e_1, \cdots, e_n$, then they also agree on the value of e_{n+1}, provided these values exist. Unfortunately, it seems that the authors have implicitly assumed that each element name in the XFD corresponds to a single node in the subtree (which is not always the case), because no exact definition of the "value" of an element is provided. There is also a strong structural restriction on the path and elements in the FD: if an element's ancestor appears in the XFD, then so must its parent. It is not hard to see that the expressive power suffered from this restriction.

Both (Arenas & Libkin 2004) and (Vincent et al. 2004) regard a simple element as having a child node labelled S, under which is attached the value of the element. The definition of XML FDs in (Arenas & Libkin 2004) is based on the so-called *tree-tuples*. Essentially, the authors treat a DTD as a single relation schema, a distinct path (the path in both (Arenas & Libkin 2004) and (Vincent et al. 2004) are of the form $r.p$ where p is a natural path) in a DTD as an attribute, and a tree-tuple a tuple in that relation. In a tree tuple, each path which ends with an element name is mapped to a distinct node or the null value ($\perp$), and every other path (ending with an attribute

name or S) is mapped to either a string (PCDATA) or $\perp$. An XML data tree T conforming to the DTD can then be regarded as consisting of a set of maximal tree tuples, here maximal means, roughly, that the tree tuple can not be extended by replacing null with non-null values while still being a subtree of T. For example, the XML tree shown in Figure 7 can be

```
<!ELEMENT school(course)* >
<!ELEMENT course (students) >
<!ATTLIST course
        cno CDATA #REQUIRED>
<!ELEMENT students (student)*>
<!ELEMENT student (name)>
<!ATTLIST student
        sno CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
```

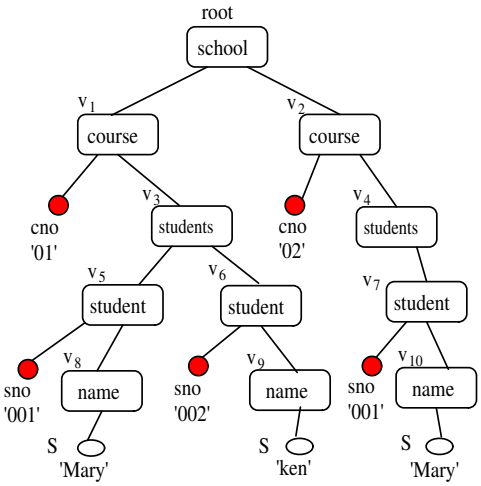

Figure 7: The Course-Student example

regarded as an instance of the relation schema with attributes

school,
school.course,
school.course.@cno,
school.course.students,
school.course.students.student,
school.course.students.student.sno,
school.course.students.student.name,
school.course.students.student.name.S,

and the instance consists of the (maximal tree) tuples

$(root,\ v_1,\ 01,\ v_3,\ v_5,\ 001,\ v_8,\ Mary)$,
$(root,\ v_1,\ 01,\ v_3,\ v_6,\ 002,\ v_9,\ Ken)$, and
$(root,\ v_1,\ 02,\ v_4,\ v_7,\ 001,\ v_{10},\ Mary)$.

An XFD is defined as $S_1 \rightarrow S_2$, where S_1 and S_2 are sets of paths. An XML document conforming to the DTD is said to satisfy the functional dependency if for every two maximal tree-tuples t_1 and t_2, and every path $p_2 \in S_2$, whenever $t_1.S_1$ is not null and $t_1.S_1 = t_2.S_1$, then $t_1.p_2 = t_2.p_2$. For example, the XML tree in Figure 7 satisfies the functional dependency

school.course.students.student.sno $\rightarrow$
school.course.students.student.name.s (**)

but not

school.course.students.student.sno $\rightarrow$
school.course.students.student.name.

(Vincent et al. 2004) defines an XFD that is close to that in (Arenas & Libkin 2004), with a few significant differences nevertheless. Rather than using DTDs, the authors consider a scheme file as a set of closed paths; rather than using a tree tuple to link the values in both sides of the FD, they use the concept of "closest node" to link the values. More specifically, an XFD is of the form

$$p_1, p_2, \cdots, p_n \rightarrow q$$

where p_i and q are paths. The satisfaction of the above XFD by an XML Tree T can be checked as follows: [Step 1] For $i \in [1, n]$, (1) find the common prefix of p_i and q, denoted $pre(p_i, q)$; (2) find the target set $root[pre(p_i, q)]$; (3) for each distinct path instance I of q, find the unique node $x_{I,i}$ which is common to I and $root[pre(p_i, q)]$; (4) if $last(p_i)$ is not an element, then find the set $N_{I,i}$ of nodes which are descendants of $x_{I,i}$ and are in the target set $root[p_i]$. Also find the set $val(N_{I,i})$ of values of the nodes in $N_{I,i}$. [Step 2] If we cannot find two distinct path instances I and J, such that for all $i \in [1, n]$, $x_{I,i} = x_{J,i}$ (when $last(p_i)$ is an element), or $val(N_{I,i}) \cup val(N_{J,i})$ contains $\perp$ or $val(N_{I,i}) \cap val(N_{J,i}) \neq \emptyset$ (when $last(p_i)$ is an attribute or S), but the value of last node in I and J are not equal, then the XNF is satisfied by T. For example, in the XML tree of Figure 7, the XFD (**) can be shown to hold as follows: [Step 1] (1) The common prefix of the two paths in (**) is $pre \equiv school.course.students.student$, and (2) the target set $root[pre] = \{v_5, v_6, v_7\}$; (3) for the path instances of the RHS $I = root.v_1.v_3.v_5.v_8.S$, $J = root.v_1.v_3.v_6.v_9.S$, and $K = root.v_2.v_4.v_7.v_{10}.S$, find the nodes $X_I = v_5$, $X_J = v_6$ and $X_K = v_7$; (4) find N_I, N_J and N_K, which are the singleton sets containing the sno node under v_5, v_6 and v_7 respectively; [Step 2] Only the values of the nodes in N_I and N_K are equal, but so are the values of last node in the corresponding path instances I and K.

A major difference between (Vincent et al. 2004) and (Arenas & Libkin 2004) lies in the treatment of null values (missing nodes). In the XML tree of Figure 7, if we remove both name values under v_5 and v_7 (or alternatively, if we remove the sno value under v_5), then the XFD (**) is still considered satisfied by (Arenas & Libkin 2004), but not by (Vincent et al. 2004). Another important difference between (Vincent et al. 2004) and (Arenas & Libkin 2004) lies in the set of trivial XFDs. Since the XFDs in (Arenas & Libkin 2004) are defined for a DTD, while the XFDs in (Vincent et al. 2004) are defined for a set of closed paths, some trivial XFDs in (Arenas & Libkin 2004) may turn out to be non-trivial in (Vincent et al. 2004). This is because a DTD puts more restrictions on conforming XML trees than a set of closed paths.

The main problem with the definitions of XFDs in (Arenas & Libkin 2004) and (Vincent et al. 2004) is that some natural constraints can not be expressed, as already seen in Section 3. In addition, since no value equality between two nodes is defined, it is sometimes cumbersome to express some functional dependencies. For instance, in the DTD below (PCDATA elements are omitted),

```
<!ELEMENT school(course)* >
<!ELEMENT course (students) >
  <!ATTLIST course
      cno CDATA #REQUIRED>
<!ELEMENT students (student)*>
<!ELEMENT student (details, grade)>
  <!ATTLIST student
          sno CDATA #REQUIRED>
<!ELEMENT details(name, address, tel)>
<!ELEMENT name (fname, lname)>
<!ELEMENT address(st, city, state, pcode)>
<!ELEMENT tel(areacode, phone, ext)
```

if we want to say student number determines student details, rather than using a simple expression $sno \rightarrow details$, we have to use either a long XFD expression that has 9 paths on the RHS or 9 separate XFDs.

The XFD of (Vincent et al. 2004) is equivalent to a special case of our XML FD except for the treatment of null values for non-element paths on the LHS. We are yet to find a meaningful example where a constraint can be expressed by the FDs in (Arenas & Libkin 2004) but not ours.

Unlike other previous work that use paths to define XFDs, (Hartmann & Link 2003) defines two types of XFDs using *homomorphism, v-subtrees* and *isomorphism* of XML trees. Homomorphism between two trees is a mapping from the nodes of one tree to another that preserves the root, label, and kind of nodes, and it is used to define conformity of an XML tree to an XML schema file (represented by a schema tree). A v-subtree is a subtree which roots at node v and it is determined by the paths from v to a subset of leaves of the original tree. The isomorphism of two subtrees is a 1-1 mapping between the two sets of nodes which is homomorphic in both directions. Isomorphism is used to define equivalence between two XML trees. The equivalence of two subtrees is similar to the value equality of their roots. An XFD is defined to be of the form $v : X \rightarrow Y$, where v is a node in the schema tree, and X and Y are v-subgraphs in the schema tree. Two types of satisfaction by a conforming XML tree are defined, and they represent two different types constraints. This allows the XFDs in (Hartmann & Link 2003) to express some constraints involving set equality (like our XFDs) as well as some constraints similar to those in (Arenas & Libkin 2004) and (Lee et al. 2002).

None of the XFDs in the other previous works takes set equality into consideration. We believe that set-equality is natural and common in real applications and should be included in defining data dependencies. Notably both (Roth, Korth & Silberschatz 1988) and (Hara & Davidson n.d.) have considered set equality in their definitions of functional dependencies in nested relations.

We are not aware of any formal definitions of EGDs for XML data, although (Lee & Wu 2000) describes a totally different type of constraints in XML, which they call equality-generating dependencies: if an element v can have at most one subelement, then when v_1 and v_2 are known to be subelements of v, they must be the same element.

8 Conclusion and Future Work

We have studied a new type of FDs as well as EGDs for data-centric XML documents, and proposed normal forms of DTDs that prevent data redundancies with respect to these dependencies. However, many issues remain to be resolved, and we plan to investigate these issues in our future work.

As an immediate task, we would like to find efficient algorithms for the implication problem and computation of the closure of our data dependencies. This must be done before we can efficiently check a DTD is in normal form. The normalization process has to be carefully designed too.

We would also like to extend our work to the design of XML documents in which the ordering of subelements is important or where there are mixed contents, because such XML documents are ubiquitous.

References

Arenas, M. & Libkin, L. (2004), 'A normal form for XML documents', *ACM Transactions on Database Systems* **29**, 195–232.

Buneman, P., Davidson, S. B., Fan, W. & Hara, C. S. (2002), 'Keys for XML', *Computer Networks* **39**(5), 473–487.

Buneman, P., Davidson, S. B., Fan, W., Hara, C. S. & Tan, W. C. (2001), 'Reasoning about keys for XML', *IDPL'2001, Lecture Notes in Computer Science* **2397**, 133–148.

Fagin, R. & Vardi, M. Y. (1984), The theory of data dependencies—an overview, *in* 'Automata, Languages and Programming, 11th Colloquium', Vol. 172 of *Lecture Notes in Computer Science*, pp. 1–22.

Fan, W., Schwenzer, P. & Wu, K. (2001), 'Keys with upward wildcards for xml', *DEXA'2001, Lecture Notes in Computer Science* **2113**, 557–567.

Hara, C. S. & Davidson, S. B. (n.d.), Reasoning about nested functional dependencies, *in* 'PODS'1999', pp. 91–100.

Hartmann, S. & Link, S. (2003), More functional dependencies for XML, *in* 'ADBIS 2003', pp. 355–369.

Lee, D. & Wu, W. W. (2000), Constraints-preserving transformation from XML to document type definition to relational schema, Technical Report UCLA-CS-TR-200001, Dept. of Computer Science, Uni of California, Los Angeles.

Lee, M. L., Ling, T. W. & Low, W. L. (2002), 'Designing functional dependencies for XML', *EDBT'2002, Lecture Notes in Computer Science* **2287**, 124–141.

Roth, M. A., Korth, H. F. & Silberschatz, A. (1988), 'Extended algebra and calculus for nested relational databases', *ACM Transactions on Database Systems* **13**(4), 389–417.

Sowa, J. F. (n.d.), Building, sharing, and merging ontologies.

Vincent, M. W. & Liu, J. (2003), Multivalued dependencies and a 4NF for XML, *in* 'CAiSE 2003', pp. 14–29.

Vincent, M. W., Liu, J. & Liu, C. (2004), 'Strong functional dependencies and their applicatiopn to normal forms in XML', *ACM Transactions on Database Systems* **29**(3), 445–462.

Wang, J. (2004), Database design using equality generating dependencies, Technical report, School of Information Technology, Griffith Univerty, Gold Coast, Australia.

Wu, X., Ling, T., Y.Lee, S., Lee, M. L. & Dobbie, G. (2002), 'NF-SS: A normal form for semistructured schema', *ER Workshop'2001, Lecture Notes in Computer Science* **2465**, 292–305.