Transformation of XML Data into XML Normal Form

Tadeusz Pankowski
Institute of Control and Information Engineering
Poznań University of Technology
Pl. M.S.-Curie 5, 60-965 Poznań, Poland
E-mail: tadeusz.pankowski@put.poznan.pl

Tomasz Piłka
Faculty of Mathematics and Computer Science
Adam Mickiewicz University
ul. Umultowska 87, 61-614 Poznań, Poland
E-mail: tomasz.pilka@amu.edu.pl

Keywords: XML, XML functional dependencies, XML normal forms, XML schema design

Received: March 4, 2009

Normalization as a way of producing good database designs is a well understood topic for relational data. In this paper we discuss the problem of eliminating redundancies and preserving data dependencies in XML data when an XML schema is normalized. Normalization is one of the main tasks in relational database design, where 3NF or BCNF, is to be reached. However, neither of them is ideal: 3NF preserves dependencies but may not always eliminate redundancies, BCNF on the contrary – always eliminates redundancies but may not preserve constraints. In this paper we consider the possibility of achieving both redundancy-free and dependency preserving form of XML schema. We show how the XML normal form can be obtained for a class of XML schemas and a class of XML functional dependencies. We relate our approach to the decomposition algorithm proposed by Arenas and Libkin in [1].

Povzetek: Razvita je metoda pretvorbe XML podatkov v normalizirano obliko s poudarkom na odstranjevanju redundanc.

1 Introduction

Normal forms, as desired forms for schemas defining structures and properties of data collections, were first proposed and investigated by Codd in early 70s. The most important of them are 3NF (*Third Normal Form*) [2] and BCND (*Boyce-Codd Normal Form*) [3]), where BCNF is a stronger definition of 3NF. Definitions of normal forms are based on the functional dependencies defined among the attributes constituting the relational schema, and specify requirements to be satisfied by the set of these functional dependencies. Then any relation that is an instance of the schema and satisfies the given set of functional dependencies, is free of harmful redundancies and anomalies. In the normalization process, the initial poor designed relational schema is decomposed into an equivalent set of well-designed schemas, i.e. into schemas in desired normal forms (usually in 3NF and often also in BCNF).

Recently, as XML becomes popular as the standard data model for storing and interchanging data on the Web and more companies adopt XML as the primary data model for storing information, XML schema design has become an increasingly important issue. Thus, we observe attempts to extend normal forms and database design principles to XML databases. Research on normalization of XML data was reported in a number of papers. In [1], Arenas and Libkin extended the relational tuple-oriented definition of functional dependencies to so-called *tree tuple*-based functional dependencies and developed the first theory of XML functional dependencies (XFDs) and XML normal forms (XNFs). This approach was further studied by Kolahi [4, 5], and Kolahi and Libkin [6]. Integrity constraints for XML data (including keys) were studied extensively in Buneman et al. in [7], and Fan and Simeon in [8]. The equivalence between XFDs and relational FDs was investigated by Vincent et al. in [9, 10]. Yu and Jagadish proposed in [11] a new XML normal form, called GTT-XNF, that is based on *Generalized Tree Tuples* (GTT).

Central objectives of a good schema design is to avoid data redundancies and to preserve dependencies enforced by the application domain (these dependencies are formalized by means of functional dependencies). Existence of redundancy can lead not only to a higher data storage cost but also to increased costs for data transfer and data manipulation. It can also lead to update anomalies [12].

One strategy to avoid data redundancies is to design redundancy-free schema. One can start from an intuitively correct XML schema and a given set of functional dependencies reflecting some rules existing in application domain. Then the schema is normalized, i.e. restructured, in such a way that the newly obtained schema has no re-

dundancy, preserves all data (is a lossless decomposition) and preserves all dependencies. In general, obtaining all of these three objectives is not always possible, as has been shown for relational schemas [13]. However, in the case of XML schema, especially thanks to its hierarchical structure, this goal can be more often achieved [4].

1.1 Related work

An algorithm, called *the Decomposition Algorithm* (DA) normalizing XML schemas was proposed in [1]. The algorithm converts any DTD, given a set of XML functional dependencies (XFDs), into DTD in XML normal form (XNF). The decomposition algorithm consists of two basic operations: *moving attributes*, and *creating new element types*. These operations are performed when an XFD violating XNF is identified. Thus, the basic idea is similar to normalization of relational data, when the second normal form (2NF), the third normal form (3NF), or the Boyce-Codd normal form (BCND) are to be achieved. In the relational counterparts during the normalization process a relational schema is decomposed into a set of its projections. Thus, we can obtain a set of separate relational schemas as the result of the normalization process performed for an initial relational schema. In the case of XML documents, the result is still one XML document restructured accordingly. In [14], an information-theoretic approach to normal forms for relational and XML data has been developed. Some other papers, e.g. [5, 15, 16, 6] study XML design and normalization for native or relational storage of XML documents. In [17, 18] approaches for obtaining well-designed XML schemas from conceptual ER (*Entity-Relationship*) schemas have been discussed.

1.2 Contribution

In this paper we describe a systematic approach to the normalization of XML data when so-called *cyclic functional dependencies exist*, i.e. dependencies, which in the relational case are functional dependencies of the form $\{A, B \rightarrow C, C \rightarrow A\}$, where A, B, C are attributes in a relational schema $R(A, B, C)$. It is well known from relational database theory [13], that the schema $R(A, B, C)$ is then in 3NF, but is not in BCNF. As a result, instances of $R(A, B, C)$ have redundancies, but decomposition the schema into BCNF leads to two schemas $R1(C, A)$ and $R2(C, B)$, which are free of redundancy but do not preserve dependency $A, B \rightarrow C$.

We focus on normalization procedure for XML schemas with cyclic XFDs. The contributions of the paper are the following:

- We use a language based on tree patterns [19, 20] to express XML schemas and XML functional dependencies. This notation is used in the formal analysis of properties of XML normal form as well as the base for developing transformation algorithms.
- We propose an approach to obtain XNF starting from ER schema. In the first step the ER schema is converted into an XML schema satisfying a *necessary conditions* (see Theorem 7.3) which is a prerequisite to successful applying of DA algorithm [1]. We show that the presence of cyclic dependencies results in a bad behavior of the DA algorithm.

This paper is organized as follows. In Section 2 we introduce a running example and motivate the research. In Section 3 a relational form of the running example is considered and some problems with its normalization are discussed. Basic notations relevant to the discussed issue from the XML perspective, are introduced in Section 4. We define tree patterns and use them to formal definition of tree tuples and instances. In Section 5, tree patterns are used to define data dependencies: XML functional dependencies (XFDs) and keys (a subclass of XFDs). In Section 6, an XML normal form (XNF) is defined (according to [1]) and we show how this form can be obtained for our running example. We discuss different normalization alternatives – we show advantages and drawbacks of some schema choices. A method for transforming an XML schema into XNF is proposed in Section 7. First, for the XML schema the conceptual model in a form of ER schema is created. This schema and functional dependencies among its attributes are the basis for creating an initial XML schema satisfying the necessary condition formulated in Theorem 7.3. This schema is the subject for further normalization. Section 8 concludes the paper.

2 Redundancies in XML data - motivation example

Our primary goal is to devise methods which will allow checking correctness of XML data and designing its expected (normalized) form. We expect such data to be devoid of the redundancies and immune to the update anomalies.

In the case of XML schemas some redundancy problems may occur because of bad design of hierarchical structure of XML document. On the other hand, the hierarchical structure of this data can sometimes help conduct the normalization of XML data.

Example 2.1. *Let us consider an XML schema tree (Figure 1) that describes a fragment of a database for storing data about parts and suppliers offering these parts. Its DTD specification is given in Figure 2, and an instance of this schema is depicted in Figure 3. Each* $part$ *element has identifier* pId*. One part may be offered by zero or more suppliers. Offers are stored in* $offer$ *elements. Each offer has: offer identifier* `oId`*, supplier identifier* `sId`*, price* `price`*, delivery time* `delivTime`*, and delivery place* `delivPlace`*.*

We assume that the following constraints must be satisfied by any instance of this schema:

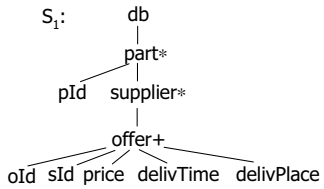

Figure 1: Sample XML schema tree

```
db        →  part*
part      →  pId supplier*
supplier  →  offer+
offer     →  oId sId price delivTime
             delivPlace
```

Figure 2: DTD productions describing the XML schema in Figure 1

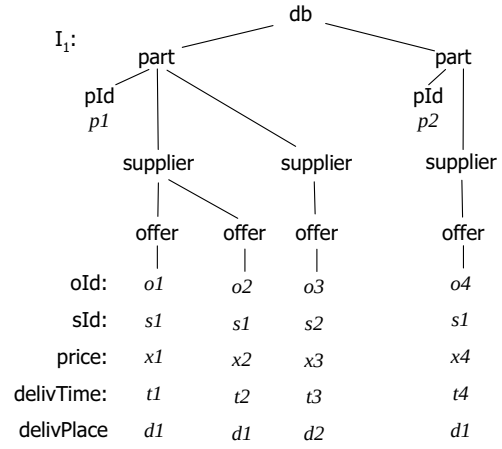

Figure 3: Sample instance of schema S_1

- all $offer$ children of the same $supplier$ must have the same values of sId; this is similar to relational functional dependencies, but now we refer to both the values (text value of sId), and to structure (children of the same $supplier$).
- $delivPlace$ functionally depends on part (pId) and supplier (sId), i.e. when a supplier has two different offers for the same part (possibly with different $delivTime$ and/or $price$) the $delivPlace$ is the same - see offers $o1$ and $o2$ in Figure 3.
- $delivPlace$ functionally determines supplier (sId). It means that having a delivery place ($delivPlace$) we exactly now which supplier is associated to this place; although one supplier can own many delivery places. For example, in Figure 3 $d1$ is delivery place uniquely associated to the supplier $s1$.

It is easily seen that schema in Figure 1 leads to redundancy: sid (and also all other data describing suppliers such as e.g.: name and address) and $delivPlace$ are stored multiple times for a supplier.

Further on we will show that a special caution should be paid to such kind of dependencies as these in which participates $delivPlace$. In this case we have to do with "cyclic" dependencies, i.e. $delivPlace$ depends on pId and sId ($pId, sId \rightarrow delivPlace$) and sId depends on $delivPlace$ ($delivPlace \rightarrow sId$).

3 Redundancies and dependencies in relational databases

3.1 Relational schemas and functional dependencies

In relational data model, a relational schema is understood as a pair $\mathcal{R} = (U, F)$, where U is a finite set of *attributes*, and F is a set of *functional dependencies* over F. A functional dependence (FD) as an expression of he form

$$X \rightarrow Y,$$

where $X, Y \subseteq U$ are subsets of U. If $Y \subseteq X$, then $X \rightarrow Y$ is a *trivial* FD. By F^+ we denote all dependencies which can be infer from F using, for example, Armstrong's axioms [21, **?**].

A *relation* of type U is a finite set of tuples of type U. Let $U = \{A_1, ..., A_n\}$ and $dom(A)$ be the *domain* of attribute $A \in U$. Then a tuple $[A_1 : a_1, ..., A_n : a_n]$, where $a_i \in dom(A_i)$, is a tuple of type U.

A relation R conforms to a schema $\mathcal{R} = (U, F)$ (is an instance of this schema) if R is of type U, and all dependencies from F^+ are satisfied by R. An FD $X \rightarrow Y$ is satisfied by R, denoted $R \models X \rightarrow Y$, if for each tuples $r_1, r_2 \in R$ holds

$$\pi_X(r_1) = \pi_X(r_2) \Rightarrow \pi_Y(r_1) = \pi_Y(r_2),$$

where $\pi_X(r)$ is the *projection* of tuple r on the set X of attributes.

A *key* in $\mathcal{R} = (U, F)$ is such a minimal set K of attributes that $K \rightarrow U$ is in F^+. Then each $A \in K$ is called a *prime* attribute.

3.2 Normalization of relational schemas

The main task in relational schema normalization is producing such a set of schemas that posses the required form, usually 3NF or BCNF. The normalization process consists in decomposition of a given input schema. The other approach consists in synthesizing 3NF from functional dependencies [22].

Ideally, a decomposition of a schema should be lossless, i.e. should preserve data and dependencies. Let $\mathcal{R} = (U, F)$, $U_1, U_2 \subseteq U$, and $U_1 \cup U_2 = U$, then schemas $\mathcal{R}_1 = (U_1, F_1)$ and $\mathcal{R}_2 = (U_2, F_2)$ are a lossless decomposition of $\mathcal{R} = (U, F)$, iff:

– The decomposition preserves data, i.e. for each instance R of $\mathcal{R}$ the natural join of projections of R on U_1 and U_2 produces the relation equal to R, i.e.

$$R = \pi_{U_1}(R) \bowtie \pi_{U_2}(R).$$

– The decomposition preserves dependencies, i.e.

$$F^+ = (F_1 \cup F_2)^+,$$

where $F_1 = \{X \to Y \mid X \to Y \in F \wedge X \cup Y \subseteq U_1\}$, and similarly for F_2.

The decomposition $((U_1, F_1), (U_2, F_2))$ of (U, F) preserves data, if $U_1 \cap U_2 \to U_1 \in F^+$ (or, symmetrically, $U_1 \cap U_2 \to U_2 \in F^+$) [23]. Then we say that the *decomposition is determined* by the functional dependence $U_1 \cap U_2 \to U_1 \in F^+$.

A schema $\mathcal{R} = (U, F)$ is in 3NF if for every FD $X \to A \in F^+$, holds:

– X is a superkey, i.e. a key is a part of X, or

– A is prime.

The second condition says that only prime attributes may be functionally dependent on a set of attributes which is not a key. A schema is in BCNF if only the first condition of the two above is allowed. It means, that if whenever a set X determines functionally an attribute A, then X is a superkey, i.e. determines the whole set U.

The aim of a normalization process is to develop normal forms by analyzing functional dependencies and successive decomposition of the input relational schema into its projections. In this way a well-designed schema can be obtained, where unnecessary redundancies and update anomalies had been eliminated. In practice, 3NF is accepted as the most desirable form of relational schemas It does not eliminate all redundancies but guaranties dependency preservation. On contrast, BCNF eliminates all redundancies but does not preserve all dependencies.

In [6] it was shown that 3NF has the least amount of redundancy among all dependency preserving normal forms. The research adopts a recently proposed information-theoretic framework for reasoning about database designs [14].

3.3 Relational analysis of XML schema

Let us consider the relational representation of the data structure presented in Figure 1. Then we have the following relational schema:
$\mathcal{R} = (U, F)$, where
$U = \{oId, sId, pId, price, delivTime, delivPlace\}$,
$F = \{oId \to sId, pId, price, delivTime, delivPlace,$
$sId, pId \to delivPlace,$
$delivPlace \to sId\}$.

In $\mathcal{R}$ there is only one *key*. The key consists of one attribute oId since all attributes in U functionally depends on oId. Thus, R is in 2NF and oId is the only prime (key) attribute in $\mathcal{R}$. Additionally, we assume that a given supplier delivers a given part exactly to one place ($pId, sId \to delivPlace$). Moreover, delivery place $delivPlace$ is connected with only one supplier ($delivPlace \to sId$).

R is not in 3NF because of the functional dependency $sId, pId \to delivPlace$:

– sId, pId is not a superkey, and

– $delivPlace$ is not a prime attribute in U.

Similarly for $delivPlace \to sId$.

In this case the lack of 3NF is the source of redundancies and update anomalies. Indeed, for example, the value of $delivPlace$ will be repeated as many times as many different tuples with the same value of the pair (sId, pId) exist in the instance of $\mathcal{R}$. To eliminate this drawback, we can decompose $\mathcal{R}$ into two relational schemas, $\mathcal{R}_1$ and $\mathcal{R}_2$, which are in 3NF. The decomposition must be based on the dependency $sId, pId \to delivPlace$ which guarantees that the decomposition preserves data. In the result we obtain:

$\mathcal{R}_1 = (U_1, F_1)$, where
$U_1 = \{oId, sId, pId, price, delivTime\}$,
$F_1 = \{oId \to sId, pId, price, delivTime\}$.

$\mathcal{R}_2 = (U_2, F_2)$, where
$U_2 = \{sId, pId, delivPlace\}$,
$F_2 = \{sId, pId \to delivPlace,$
$delivPlace \to sId\}$.

The discussed decomposition is both data and dependencies preserving, since:
$R(U) = \pi_{U_1}(R) \bowtie \pi_{U_2}(R)$,
for every instance R of schema $\mathcal{R}$, and $F = (F_1 \cup F_2)^+$.
However, we see that $\mathcal{R}_2$ is not in BCNF, since $delivPlace$ is not a superkey in $\mathcal{R}_2$.

The lack of BCNF in $\mathcal{R}_2$ is the reason of redundancies. For example, in table R_2 we have as many duplicates of sId as many tuples with the same value of $delivPlace$ exist in this table.

R_2

sId	pId	$delivPlace$
s1	p1	d1
s1	p2	d1
s1	p3	d2
s2	p1	d3

We can further decompose $\mathcal{R}_2$ into BCNF schemas $\mathcal{R}_{21}$ and $\mathcal{R}_{22}$, taking $delivPlace \to sId$ as the base for the decomposition. Then we obtain:

$\mathcal{R}_{21} = (U_{21}, F_{21})$, where
$U_{21} = \{delivPlace, sid\}$,
$F_{21} = \{delivPlace \rightarrow sId\}$.

$\mathcal{R}_{22} = (U_{22}, F_{22})$, where
$U_{22} = \{pId, delivPlace\}$,
$F_{22} = \emptyset$.

After applying this decomposition to R_2 we obtain tables R_{21} and R_{22}:

R_{21}

sId	$delivPlace$
s1	d1
s1	d2
s2	d3

R_{22}

pId	$delivPlace$
p1	d1
p2	d1
p3	d2
p1	d3

This decomposition is information preserving, i.e.

$$R_2 = R_{21} \bowtie R_{22},$$

but does not preserve functional dependencies, i.e.

$F_2 \neq (F_{21} \cup F_{22})^+ = F_{21}$.

We can observe some negative consequences of the loss of functional dependencies in the result decomposition.

Assume that we insert the tuple $(p1, d2)$ into R_{22}. The tuple will be inserted because it does not violate any constrain imposed on $\mathcal{R}_{22}$. However, taking into account table R_{21} we see that supplier $s1$ (determined by $d2$ in force of $delivPlace \rightarrow sId$) offers part $p1$ in the place $d1$. Thus, the considered insertion violates functional dependency $sId, pId \rightarrow delivPlace$ defined in $\mathcal{R}_2$.

The considered example shows that in the case of relational databases we are not able to completely eliminate redundancies and also preserve all functional dependencies. It turns out ([6]) that the best form for relation schema is 3NF, although some redundancies in tables having this form can still remain.

In next section we will show that the hierarchical structure of XML documents can be used to overcome some of the limitations of relational normal forms [24]. As it was shown in [5], there are decompositions of XML schemas that are both information and dependency preserving. In particular, we can obtain a form of XML schema that is equivalent to BCNF, i.e. eliminates all redundancies, and additionally preserves all XML functional dependencies.

4 XML schemas and instances

Schemas for XML data are usually specified by DTD or XSD [25, 26]. In this paper an *XML schema* (a schema for short) will be specified by means of a slightly simplified version of DTD. We will assume that both attributes and elements of type `#PCDATA` will be represented by so called *terminal elements* labeled with *terminal labels* and having text values. Additionally, terminal elements may have only one occurrence within children of one non-terminal element.

Definition 4.1. *Let L be a set of labels, $Ter \subset L$ be a set of terminal labels and $r \in L - Ter$ be a root label. Let ρ be a set of productions of the form:*

$$l \rightarrow \alpha,$$

where:

- $l \in L - Ter$;
- *r does not occur on the right-hand side of any production;*
- *α is a regular expression over $L - \{r\}$ defined as follows:*
  $\alpha ::= \beta \mid \gamma$
  $\beta ::= l \mid \beta? \mid \beta* \mid \beta+$
  $\gamma ::= A \mid A? \mid \beta \mid \gamma\,\gamma \mid \gamma|\gamma$
  $l \in L - Ter - \{r\}$, $A \in Ter$.

Then the quadruple

$$S = (r, L, Ter, \rho),$$

is an XML schema.

XML schemas will be often represented as XML schema trees (Figure 1) (or as XML schema graphs when the schema is recursive). In this paper we restrict ourselves to non-recursive schemas. An example of XML schema, corresponding to the schema tree in Figure 1, was given in Figure 2, where: db is the root, $part, supplier$, and $offer$ are non-terminal labels, and $pid, oid, sid, price, delivTime, delivPlace$ are terminal labels.

The following notion of *tree patterns* [19] will be useful to define tree tuples, tree formulas and XML functional dependencies.

Definition 4.2. *Let L be a set of labels, and $Ter \subset L$ be its subset of terminal labels. A tree pattern is an expression conforming to the syntax:*

$$e ::= A \mid l \mid l/e \mid l[e_1, \ldots, e_k] \mid l[e_1, \ldots, e_k]/e,$$

where $A \in Ter$, $l \in L - Ter$, $n \leq 1$.

To denote that a tree pattern ϕ is build using the set $\{A_1, \ldots, A_n\}$ of terminal labels, we will write $\phi(A_1, \ldots, A_n)$.

Definition 4.3. *Let $\phi(A_1, \ldots, A_n)$ be a tree pattern, and $x_1, \ldots, x_n$ be text-valued variables. Then the expression*

$$\phi(A_1 : x_1, \ldots, A_n : x_n),$$

is a tree formula.

Definition 4.4. *Let $\phi(A_1 : x_1, \ldots, A_n : x_n)$ be a tree formula, and ω be a variable valuation, i.e. a function*

$$\omega : \{x_1, \ldots, x_n\} \to \texttt{Str} \cup \{\bot\},$$

where Str *is a set of text values, and $\bot$ is a distinguished null value. Then*

$t = \phi(A_1 : x_1, \ldots, A_n : x_n)(\omega) = \phi(A_1 : \omega(x_1), \ldots, A_n : \omega(x_n))$

is called a tree tuple of type $\phi(A_1, \ldots, A_n)$ with values $(A_1 : \omega(x_1), \ldots, A_n : \omega(x_n)$.

Without loss of generality, a tree tuple of type $\phi(A_1, \ldots, A_n)$ will be denoted also as:

$t = \phi(A_1 : a_1, \ldots, A_n : a_n)$,
$t = \phi(A_1, \ldots, A_n)(\omega)$,
$t = \phi(\omega(A_1), \ldots, \omega(A_n))$,
$t = \phi(A_1 : t.A_1, \ldots, A_n : t.A_n)$.

Definition 4.5. *Let t be a tree tuple of type $\phi(U)$, and let $U' = \{A_1, \ldots, A_n\} \subseteq U$. The projection of t on U', denoted $\pi_{U'}(t)$, is the set of fields $(A_i : t.A_i)$ occurring in t, i.e.:*

$$\pi_{U'}(t) = \{A_1 : t.A_1, \ldots, A_n : t.A_n\}.$$

We assume that there is a function $type(t)$ that for a tree tuple t returns its type, it will be denoted by

$$type(t) = \phi(A_1, \ldots, A_n).$$

An XML database consists of a set of XML data, and an XML data is an instance of an XML schema. It will be convenient to distinguish between two kinds of instances:

- an instance as a set of tree tuples, referred to as a *canonical instance*,
- an instance as a labeled tree, referred as an XML tree.

For one canonical instance there may be a number of XML trees representing it. In the set of all such XML trees we will be interested in such that have a required form, so called XML normal form (XNF). The canonical instance as well as all corresponding XML trees must conform to the given XML schema.

Definition 4.6. *The set of tree tuples*

$$D = \{t_1, \ldots, t_N\},$$

is a canonical instance of an XML schema $S = (r, L, Ter, \rho)$ if the type, $type(t)$, of any tuple $t \in D$ conforms to S.

The conformance of a tree tuple type to an XML schema is defined as follows.

Definition 4.7. *Let $\phi(A_1, \ldots, A_n)$ be the type of a tree tuple t. This tree pattern conforms to the XML schema $S = (r, L, Ter, \rho)$, if:*

1. *$\phi(A_1, \ldots, A_n) = r[e]$, and*
2. *e conforms to ρ according to r. The conformance of a pattern e to the set of productions ρ according to a label l, is defined as follows:*
   - *if e is A_i, then $A_i \in Ter$, and there is such the production $l \to \alpha$ in ρ that A_i occurs in α;*
   - *if e is l'/e', then there is such the production $l \to \alpha$ in ρ that l' occurs in α, and e' conforms to ρ according to l';*
   - *if e is $l'[e_1, \ldots, e_n]$, then there is such the production $l \to \alpha$ in ρ that l' occurs in α, and every pattern e_i, $1 \leq i \leq n$, conforms to ρ according to l'.*
   - *if e is $l'[e_1, \ldots, e_n]/e'$, then $l'[e_1, \ldots, e_n]$ must conform to ρ according to l, and e' must conform to ρ according to l'.*

We define XML tree as an ordered rooted node-labeled tree over a set L of labels, and a set $\texttt{Str} \cup \{\bot\}$, which elements are used as values of terminal nodes.

Definition 4.8. *An XML tree I is a tuple $(root, N^e, N^t, child, \lambda, \nu)$, where:*

- *$root$ is a distinguished root node, N^e is a finite set of non-terminal element nodes, and N^t is a finite set of terminal nodes;*
- *$child \subseteq (\{r\} \cup N^e) \times (N^e \cup N^t)$ – a relation introducing tree structure into the set $\{r\} \cup N^e \cup N^t$, where r is the root, each non-terminal node has at least one child, terminal nodes are leaves;*
- *$\lambda : \{root\} \cup N^e \cup N^t \to L$ – a function labeling nodes with labels;*
- *$\nu : N^t \to Str \cup \{\bot\}$ – a function assigning terminal nodes with values.*

Definition 4.9. *We say that an XML tree $I = (root, N^e, N^t, child, \lambda, \nu)$ conforms to an XML schema $S = (r, L, Ter, \rho)$, denoted $I \models S$, if*

- *$\lambda(root) = r$; if $n \in N^e$, then $\lambda(n) \in L - Ter$; if $n \in N^t$, then $\lambda(n) \in Ter$;*
- *if $\lambda(n) = l$, $l \to \alpha \in \rho$, and $n_1, ..., n_k$ are children of n, then the sequence $\lambda(n_1) \cdots \lambda(n_k)$ is a word in the language generated by α.*

It will be useful to perceive an XML canonical instance D with tuples of type ϕ as a pair (ϕ, Ω) (called a *description*), where Ω is a set of valuations for ϕ.

Example 4.1. *For the instance I_1 in Figure 3 we have:*
$(\phi(pId, oId, sId, price, delivTime, delivPlace)$
$\{(p1, o1, s1, x1, t1, d1), (p1, o2, s1, x2, t2, d1),$
$(p1, o3, s2, x3, t3, d2), (p2, o4, s1, x4, t4, d1)\})$.

An XML tree I satisfies a description (ϕ, Ω), if the $root$ of I satisfies ϕ for every valuation $\omega \in \Omega$, where:

1. $(I, root) \models (r[e], \omega)$, iff $\exists n \in N^e\ child(r, n) \wedge (I, n) \models (e, \omega)$;
2. $(I, n) \models (A, \omega)$, iff $\lambda(n) = A$ and $\nu(n) = \omega(A)$.
3. $(I, n) \models (l/e, \omega)$, iff $\lambda(n) = l$ and $\exists n' \in N^e$ $child(n, n') \wedge (I, n') \models (e, \omega)$
4. $(I, n) \models (l[e_1, ..., e_k], \omega)$, iff $\lambda(n) = l$ and for each i, $1 \leq i \leq k$, exists n_i such that $child(n, n_i) \wedge (I, n_i) \models (e_i, \omega))$;
5. $(I, n) \models (l[e_1, ..., e_k]/e, \omega)$, iff $(I, n) \models (l[e_1, ..., e_k], \omega)$ and exists n' such that $child(n, n') \wedge (I, n') \models (e, \omega))$.

A description (ϕ, Ω) represents a class of ϕ instances with the same set of values (the same Ω), since elements in instance trees can be grouped and nested in different ways.

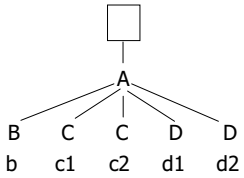

Figure 4: A simple XML tree

For example, the XML tree in Figure 4 satisfies (among others) the following two descriptions (ϕ_1, Ω_1), and (ϕ_2, Ω_2), where:

$\phi_1(B, C) = \quad /A[B, C],$
$\Omega_1 = \quad \{(b, c1), (b, c2)\};$
$\phi_2(B, C, D) = \quad /A[B, C, D],$
$\Omega_2 = \quad \{(b, c1, d1), (b, c2, d1)\}.$

5 XML functional dependencies and keys

Over an XML schema we can define some constraints, which specify functional dependencies between values and/or nodes in instances of the schema. These constraints are called XML functional dependencies (XFD).

Definition 5.1. *A tree pattern of the form $\phi(A_1, \ldots, A_k)/A$ conforming to an XML schema $S = (r, L, Ter, \rho)$ is an XML functional dependency (XFD) over S. Then we say that the set of paths terminating in $(A_1, \ldots, A_k)$ and satisfying ϕ, determines the path terminating in A and satisfying ϕ/A.*

Satisfaction of an XFD is defined against an canonical instance of XML schema.

Definition 5.2. *Let S be an XML schema, D be a canonical instance of S and $f = \phi(A_1, \ldots, A_k)/A$ be an XFD on S. We say the D satisfies f, denoted $D \models f$ if for any two tree tuples $t_1, t_2 \in D$ the following implication holds*

$$\pi_{A_1,\ldots,A_k}(t_1) = \pi_{A_1,\ldots,A_k}(t_2) \Rightarrow \pi_A(t_1) = \pi_A(t_2).$$

Assume that $A_1, \ldots, A_k, A$ terminates, respectively, paths $p_1, \ldots, p_n, p$. Then the XPath-oriented XFD $\phi(A_1, \ldots, A_k)/A$ corresponds to the following path-oriented XFD defined in [1]:

$$p_1.A_1, \ldots, p_k.A_k \rightarrow p.A.$$

The XPath-oriented definition has the following advantages:

- it is easy to check, using only the XPath semantics, whether an XML tree satisfies the XFD or not (see below),
- this form of XFD can be used to generate an XQury program performing some transformation operations (see the next section).

An XFD f can be interpreted as XPath expression, i.e. $f(x_1, \ldots, x_k) := \phi(A_1 : x_1, \ldots, A_k : x_k)/A$, that for a given valuation ω of its variables returns a sequence of objects (nodes or text values).

An XML tree $I = (S, \Omega)$ satisfies an XFD $f(\mathbf{x})$ if for each valuation $\omega \in \Omega$ of its variables, $f(\omega(\mathbf{x}))$ evaluated on I returns an empty sequence or a singleton , i.e.

$$count([\![f(\omega(\mathbf{x}))(I)]\!]) \leq 1,$$

where $[\![expr]\!](I)$ is the result of evaluation $expr$ on the instance I.

An XML tree $I = (S, \Omega)$ satisfies an XFD f if for each valuation $\omega \in \Omega$, $f(\omega)$ evaluated on I returns an empty sequence or a singleton, i.e.

$$count([\![f(\omega)(I)]\!]) \leq 1,$$

where $[\![expr]\!](I)$ is the result of evaluation $expr$ on the instance I.

Example 5.1. *Over S_1 the following XFDs can be defined:*

$f_1(oid) = /db/part[supplier/offer/oId],$
$f_2(oid) = /db/part[supplier/offer/oId]/pId,$
$f_3(pid, sid) = /db/part[pId]/supplier/$
$\quad offer[sId]/delivPlace,$
$f_4(delivPlace) = /db/part/supplier/offer[$
$\quad delivPlace]/sId,$
$f_5(pid, sid) = /db/part[pId]/supplier[offer/sId].$

According to XPath semantics [27] the expression $f_1(oid)$ by a valuation ω, is evaluated against the instance I_1 (Figure 3) as follows: (1) first, a sequence of nodes of type $/db/part$ is chosen; (2) next, for each selected node the predicate $[supplier/offer/oId = \omega(oid)]$ is tested, this predicate is true in a node n, if there exists a path of type $supplier/offer/oId$ in I_1 leading from n to a text node with the value $\omega(oid)$. We see that $count([\![f_1(\omega(oid))]\!])$ equals 1 for all four valuations satisfied by I_1, i.e. for $oid \mapsto o_1$, $oid \mapsto o_2$, $oid \mapsto o_3$, and $oid \mapsto o_4$.

Similarly, execution of $f_2(oid)(I)$ gives a singleton for any valuation of oid. These singletons are text values of the path $/db/part/pId$, where only nodes satisfying the predicate $[supplier/offer/oId = \omega(oid)]$ are taken from the set of nodes determined by $/db/part$. So, also this XFD is satisfied by I_1.

However, none of the following XFDs is satisfied in I_1:
$g_1(sid) = /db/part[supplier/offer/sId]$,
$g_2(pid) = /db/part[pId]/supplier/offer/sId$
$g_3(pid) = /db/part[pId]/supplier/offer$
$g_4(pid, sid) = /db/part[pId]/supplier/offer[sId]$,
$g_5(delivPlace) = /db/part/pId/supplier/$
$offer[delivPlace]$.

Evaluating the above XFDs against I_1, we obtain, for example:
$count([\![g_1(sid)([sid \mapsto s1])(I_1)]\!]) = 2$,
$count([\![g_2(pid)([pid \mapsto p1])]\!]) = 2$,
$count([\![g_3(pid)([pid \mapsto p1])]\!]) = 3$.

An XFD can determine functional relationship between a tuple of text values of a given tuple of paths and a path denoting either a text value (e.g. $f_2(oid)$) or a subtree (a node being the root of the subtree) (e.g. $f_1(oid)$) Ű the latter XFDs will be referred to as *XML keys*.

Definition 5.3. *A functional dependence $f = \phi(A_1, \ldots, A_k)$, where f is of type q/l and l is a non-terminal label, is an XML key for subtrees of the type q/l.*

Another notation for XML keys has been proposed in [7]. In that notation

$$(db.part, \{pId\})$$

is an *absolute key* saying that a subtree of type $db/part$ is uniquely determined by the path $db/part/pid$ (this constraint holds in the instance I_1 in Figure 3). In our XPath-oriented notation this key is the following XFD:

$$db/part[pid].$$

An example of a *relative key* ([7]) is

$$(db.part, (supplier, \{offer.sId\})),$$

that says that that in the context of $db/part$, a tree $supplier$ is determined by the path $offer/sId$. In our XPath-oriented notation this key is expressed as follows:

$$db/part[pid]/supplier[sid].$$

6 Normal form for XML

To eliminate redundancies in XML documents, some normal forms (XNF) for XML schemas have been proposed [1, 24, 4, 5]. In this paper we will define XNF in the spirit of BCNF defined for relational schemas. This approach was also used in [1].

Definition 6.1. *Let S be an XML schema and Σ be a set of XFDs defined over S. (S, Σ) is in XNF iff for any XFD $\phi(A_1, \ldots, A_k)/A \in (S, \Sigma)^+$, also $\phi(A_1, \ldots, A_k) \in (S, \Sigma)^+$,where $(S, \Sigma)^+$ is a set of XFDs being consequences of (S, Σ).*

In other word, if $\phi(A_1, \ldots, A_k)/A$ is an XFD for $q/l/A$, then $\phi(A_1, \ldots, A_k)$ is a key for q/l. It means that there is at most one subtree of type q/l for any different valuation of $(A_1, \ldots, A_k)$ in ϕ, if the value of a child A of q/l is determined by this valuation. In this way the redundancy, i.e. repetition of values in different subtrees of type q/l, is eliminated.

Let us consider the schema S_2 in Figure 5 and its instance I_2 in Figure 6.

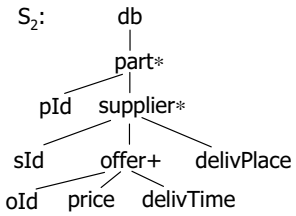

Figure 5: Restructured form of schema in Figure 1

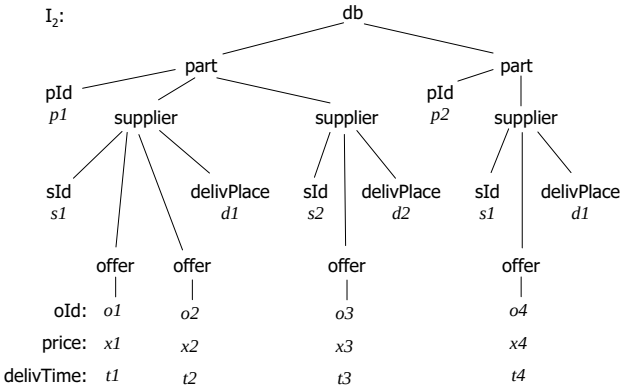

Figure 6: Instance of schema in Figure 5

The following XFD over S_2

$$/db/part/supplier[delivPlace]/sId$$

says that *delivery place* ($delivPlace$) determines the supplier (sId). However, S_2 is not in XNF, since its instance I_2 does not satisfy the key

$$/db/part/supplier[delivPlace].$$

It means that I_2 (Figure 6) is not free of redundancy (there are two different subtrees of type $supplier$ describing the same supplier, i.e. its possible name, address, etc.).

In the case of schema S_3 (Figure 7) the corresponding XFD and the key are:

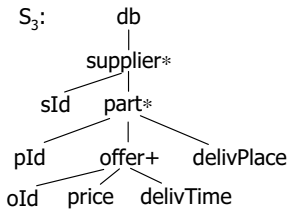

Figure 7: Restructured form of schema in Figure 1

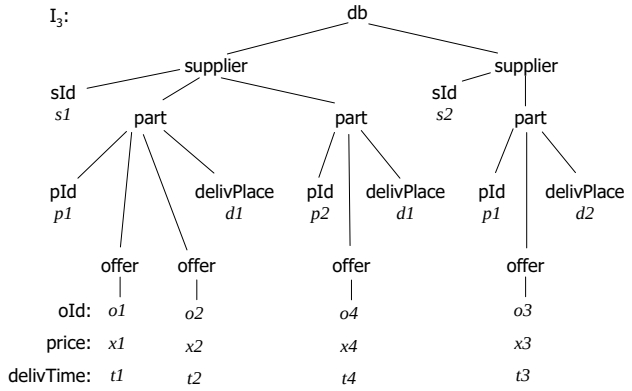

Figure 8: Instance of schema in Figure 7

$db/supplier[part/delivPlace]/sId$
$db/supplier[part/delivPlace]$.

These constraints are satisfied by I_3 (Figure 8). We see that this time, there is only one subtree of type $supplier$ for any value of $delivPlace$.

The other dependency of interest is $sId, pId \rightarrow delivPlace$. Its specification with respect to S_2 and S_3 is as follows:

$$db/part[pId]/supplier[sId]/delivPlace,$$

and

$$db/supplier[sId]/part[pId]/delivPlace.$$

It is easy to see then if these XFDs hold, then also keys

$$db/part[pId]/supplier[sId],$$

and

$$db/supplier[sId]/part[pId]$$

are satisfied.

However, neither S_2 nor S_3 is in XNF. We have already shown that there is redundancy in instances of S_2. Similarly, we see that also in instances of S_3 redundancies may occur. Indeed, since one part may be delivered by many suppliers then the description of one part may be multiplied under each supplier delivering this part, so such data as *part name*, *type*, *manufacturer* etc. will be stored many times. It results from the fact that satisfaction of $db/supplier/part[pid]/pname$ would not imply the satisfaction of $db/supplier/part[pid]$.

In Figure 9 there is schema S_4 that is in XNF. To make the example more illustrative, we added node $name$ to $part$ data. Also the instance in Figure 10 was slightly extended as compared to instances I_2 and I_3.

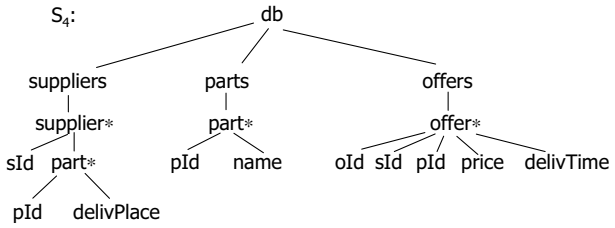

Figure 9: XNF schema for schemas S_1, S_2, and S_3

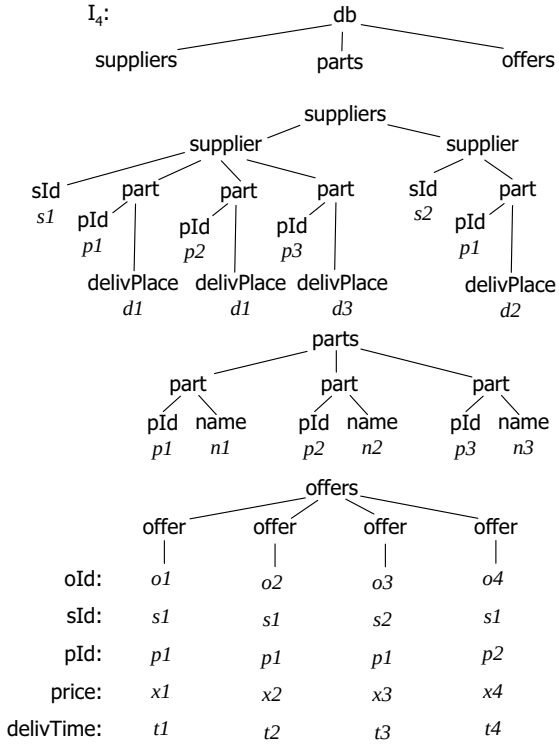

Figure 10: Instance of schema in Figure 9

XSD (XML Schema Definition) for S_4 in notation proposed in [26] is shown in Figure 11.

Note that we cannot use DTD since there are two subtrees labeled $part$, where each of them has different type: the $part$ subtree under $supplier$ consists of pId and $delivPlace$, whereas the $part$ subtree under $parts$ consists of pId and $name$. Recall that in the case of DTD each non-terminal symbol (label) can have only one type (definition), i.e. can appear on the left-hand side of exactly one production rule [26]. This difficulty might be overcame by introducing new labels (for example *partDesc*) for full descriptions of a parts.

It can be shown that S_4 satisfies the condition of XNF. Thus, this schema is both redundant-free and dependency

$$
\begin{array}{lcl}
db & \rightarrow & \mathrm{db}[content] \\
content & \rightarrow & \mathrm{suppliers}[suppliers], \mathrm{parts}[parts], \\
 & & \quad \mathrm{offers}[offers] \\
suppliers & \rightarrow & \mathrm{supplier}[supplier]* \\
parts & \rightarrow & \mathrm{parts}[part_1]* \\
offers & \rightarrow & \mathrm{offers}[offer]* \\
supplier & \rightarrow & \mathrm{sId}, \mathrm{part}[part_2] \\
part_2 & \rightarrow & \mathrm{pId}, \mathrm{delivPlace} \\
part_1 & \rightarrow & \mathrm{pId}, \mathrm{name} \\
offer & \rightarrow & \mathrm{oId}, \mathrm{sId}, \\
 & & \quad \mathrm{pId}, \mathrm{price}, \mathrm{delivTime}
\end{array}
$$

Figure 11: An XSD describing the XML schema in Figure 9

preserving. However, as we will show in the next section, schema S_4 is not the best XNF for the considered running example.

7 Transforming XML schemas to XML normal form

In the previous section we discussed an example of transforming an XML schema into XNF. We started with the schema S_1 in Figure 1, and the final schema was S_4 in Figure 9. However, the final schema has been created in a rather intuitive way. Thus, although it is in XNF it is not clear whether there exists another XNF reformulation of S_1, maybe better than X_4, or not. A systematic algorithm (the Decomposition Algorithm, DA) for normalizing an XML schema was proposed in [1]. If we apply DA to S_1, we obtain a schema that is worse than S_4. It is so due to the following reasons:

1. In DA we always obtain a normal form *relative* to a *context path*, i.e. the XNF is restricted to the subtree determined by this context path. Only if the context path is equal to the *root label* (db in our case), the XNF is *absolute*.

2. In DA it is not possible to change ordering of elements on a path, because we can only move and discard attributes or create new element types [1]. For example, the $part$ element precedes $supplier$ (is above it) in S_1 and in the result schema this ordering must remain unchanged. But then the absolute XNF for S_1 cannot be achieved.

3. The result of normalization strongly depends on the starting XML schema.

7.1 Transforming ER model to XNF

In this section we will discuss how to transform an ER (*Entity-Relationship*) schema [28] into XML schema in XML normal form (XNF). Our method is based on analyzing functional dependencies among attributes of entities involved in the ER schema.

In Figure 12 there is an ER schema corresponding to the XML schema considered in previous sections (see S_1 in Figure 1) with two additional attributes $sname$ (supplier name) and $pname$ (part name). $delivPlace$ is treated as an entity with the key attribute did.

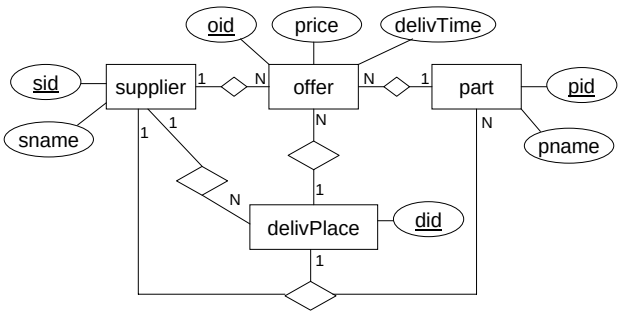

Figure 12: ER schema corresponding to S_1

The following functional dependencies are captured by the ER schema in Figure 12:

$$
\begin{array}{l}
oid \rightarrow sid, pid, did \\
sid, pid \rightarrow did \\
did \rightarrow sid \\
sid \rightarrow sname \\
pid \rightarrow pname \\
oid \rightarrow price, delivTime
\end{array}
\tag{1}
$$

The first three of them are of particular importance because they state constraints between entities (key attributes of entities).

We will proceed in two steps:

1. An initial XML schema is created using the entities names from the ER schema, their attributes and functional dependencies between key attributes. The initial schema must follow the *necessary condition* formulated in Theorem 7.3. Satisfaction of this condition is the prerequisite to obtain an XML schema in absolute XNF in the next step.

2. The DA algorithm [1] is applied to obtain the final XNF schema. In fact, only the step called *Creating New Element Types* from this algorithm is to be applied. In this way all violations caused by functional dependencies over non-key attributes (appearing on the right-hand sides of these dependencies) are resolved. Such functional dependencies in our example are for example the three last in (1).

Definition 7.1. *Let $\phi(A_1, \ldots, A_n)/B$ be an XFD of type $q/l/B$ over an XML schema S. S is called* XNF-consistent *with $\phi(A_1, \ldots, A_n)/B$, if for any instance I of S holds the implication:*

$$
\begin{array}{l}
I \models \phi(A_1, \ldots, A_n)/B \Rightarrow \\
merge_{q/l}(I) \models \phi(A_1, \ldots, A_n),
\end{array}
$$

where $merge_{q/l}(I)$ is the result of merging subtrees of type q/l in I.

Let I be an XML tree and $T = (r, \tau)$ be a subtree of type q/l of I, i.e. T belongs to the result of evaluation of the path expression q/l on the instance I, $T \in [\![q/l(I)]\!]$. Then r is a list of the form $r := (A_1 : a_1, \ldots, A_n : a_n)$, and τ is a list of subtrees (each of these lists may be empty).

Definition 7.2. *Two subtrees $T_1 = (r_1, \tau_1)$, and $T_2 = (r_2, \tau_2)$ of type q/l of an instance I are joinable if $r_1 = r_2$, and then:*

- *$(r, \tau_1) \bowtie (r, \tau_2) = (r, \tau_1||\tau_2)$, where $\tau_1||\tau_2$ is concatenation of lists τ_1 and τ_2;*
- *the merge operation on all subtrees of type q/l of an instance I is defined as follows:*
  $merge_{q/l}(I) =$
  for each $T_1, T_2 \in [\![q/l(I)]\!]$
  if T_1 **and** T_2 are joinable **then**
  $I := I - q/l[T_1] - q/l[T_2] \cup q/l[T_1 \bowtie T_2]$.

Example 7.1. *Let S be defined by*
$l \to l_1*$
$l_1 \to A_1\ B\ l_2*$
$l_2 \to A_2\ C$
and let terminal labels A_1, A_2 functionally determine B, i.e. $A_1, A_2 \to B$. Then the corresponding XFD is

$$\phi(A_1, A_2)/B := l/l_1[A_1, l_2[A_2]]/B.$$

We see that I (Figure 13) satisfies $\phi(A_1, A_2)/B$, i.e. for the values of the pair of paths $(l/l_1/A_1, l/l_1/l_2/A_2)$ the value of the path $l/l_1/B$ is uniquely determined. Similarly, the merged form of I, $merge_{l/l_1}(I)$, satisfies $\phi(A_1, A_2)$, i.e. the pair $(l/l_1/A_1, l/l_1/l_2/A_2)$ of path values uniquely determines the node of type l/l_1. Note that I in its unmerged form does not satisfy $\phi(A_1, A_2)$, because there are two nodes of type l/l_1 corresponding to the same pair of path values of the type $(l/l_1/A_1, l/l_1/l_2/A_2)$.

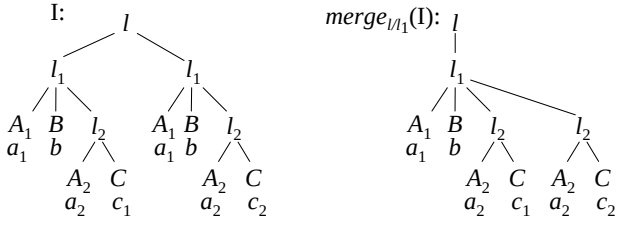

Figure 13: Instance I of the schema from Example 7.1 and its merged form

The following theorem formulates the necessary condition for XML schema to be XNF-consistent with an XDF defined over this schema.

Theorem 7.3. *Let $f := \phi(A_1, \ldots, A_n)/B$ be an XFD of type $q/l/B$ over an XML schema S. Then S is XNF-consistent with f if the set consisting of labels of all terminal children of q/l is functionally dependent on the set of terminal labels occurring in f.*

Proof. We will proof the theorem by contradiction. Let us assume that there exists a terminal child of l labeled C and C is not functionally dependent on the set $\{A_1, \ldots, A_n\}$ of terminal labels occurring in f. Then the tree of the form
$I = \phi(A_1 : a_1, \ldots, A_n : a_n)$
$\cup\{q/l[B : b, C : c_1], q/l[B : b, C : c_2]\}$,
is a subtree of an instance of S in which $\phi(A_1, \ldots, A_n)/B$ is satisfied. However, this subtree violates $\phi(A_1, \ldots, A_n)$. This violation follows from the fact that there are two subtrees rooted in q/l, one with terminal children $(B : b, C : c_1)$, and the other with terminal children $(B : b, C : c_2)$, so this subtrees cannot be merged. Thus S is not XDF-consistent with f. □

In the schema in the following example the necessary condition formulated in Theorem 7.3 does not hold.

Example 7.2. *Let S be defined by*
$l \to l_1*$
$l_1 \to A_1\ l_2*$
$l_2 \to A_2\ B\ C$
and let terminal labels A_1, A_2 functionally determine B, i.e. $A_1, A_2 \to B$. Then the corresponding XFD is

$$\phi(A_1, A_2)/B := l/l_1[A_1]/l_2[A_2]/B.$$

Assume that C does not depend on $\{A_1, A_2\}$. Schema S is not XNF-consistent with $\phi(A_1, A_2)/B$, since we have (see Figure 14):
$I = l/l_1(A_1 : a_1)/l_2(A_2 : a_2)$
$\cup\{l/l_1/l_2((B : b, C : c_1), (B : b, C : c_2))\}$
$= merge_{l/l_1/l_2}(I)$
Thus, I violates $\phi(A_1, A_2)$ before and after application of the merging operation, in both cases there are two nodes of type $l/l_1/l_2$ corresponding to the same pair of values (a_1, a_2) of the pair of paths $(l/l_1/A_1, l/l_1/l_2/A_2)$.

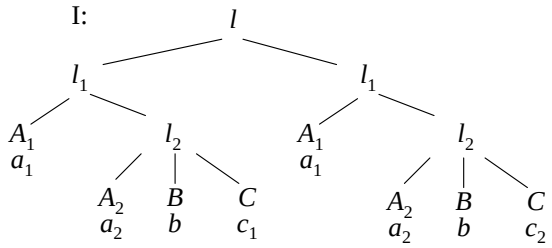

Figure 14: Instance of the schema from Example 7.2 that satisfies $\phi(A_1, A_2)/B$ and violates $\phi(A_1, A_2)$

Now, using the Theorem 7.3 and DA, the transformation of ER into XNF is realized in the following two steps:

1. We start with functional dependencies defined over key attributes of entities modeled by ER schema. Following the requirements of Theorem 7.3 we obtain the

following DTD for ER schema in Figure 12:

$$\begin{array}{l} db \rightarrow supplier* \\ supplier \rightarrow sid\ \ sname\ \ part* \\ part \rightarrow pid\ \ pname\ \ did\ \ offer* \\ offer \rightarrow oid\ \ price\ \ delivTime \end{array} \quad (2)$$

Now, the remaining functional dependencies specified in (1) must be taken into account.

2. The DTD obtained in the first step is the subject of the decomposition by means of the DA algorithm. It is easily seen that the XFD corresponding to $pid \rightarrow pname$ is anomalous. The step *Creating new element types* of DA converts (2) into

$$\begin{array}{l} db \rightarrow supplier*\ \ partDesc* \\ supplier \rightarrow sid\ \ sname\ \ part* \\ partDesc \rightarrow pid\ \ pname \\ part \rightarrow pid\ \ did\ \ offer* \\ offer \rightarrow oid\ \ price\ \ delivTime \end{array} \quad (3)$$

Applying the above steps to the ER schema from Figure 12 gives the XML schema in XNF depicted in Figure 15 and its instance in Figure 16.

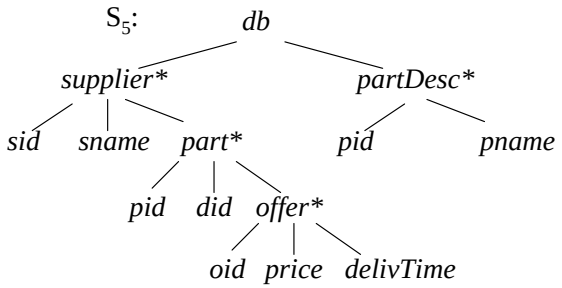

Figure 15: The result of transformation ER schema from Figure 12 to XML schema in XNF

8 Conclusion

In this paper, we discussed how the concept of database normalization can be used in the case of XML data. Normalization is commonly used to develop a relational schema free of unnecessary redundancies and preserving all data dependencies existing in application domain. In order to apply this approach to design XML schemas, we introduced a language for expressing XML functional dependencies. In fact, this language is a class of XPath expressions, so its syntax and semantics are defined precisely. We define the notion of satisfaction of XML functional dependence by an XML tree. To define XNF we use the approach proposed in [24].

All considerations are illustrated by the running example. We discuss various issues connected with normalization and compare them with issues faced in the case of relational databases. We show how to develop redundancy-free and dependency preserving XML schema. It is worth mentioning that the relational version of the schema cannot be structured in redundancy-free and dependency preserving form. In this case, preservation of all dependencies requires 3NF but then some redundancy is present. Further normalization to BCNF eliminates redundancies but does not preserve dependencies. In the case of XML, thanks to its hierarchical nature, we can achieve both properties. However, it is not clear if this is true in all cases (see e.g. [5].

We have proposed a method for normalizing XML data in to steps. First, we build a conceptual model by means of ER schema and specify all functional dependencies among its attributes. Following the necessary condition formulated in Theorem 7.3, an initial XML schema is created. This schema is XNF-consistent with all XML functional dependencies under consideration. Such the schema can be further normalized, for example using the decomposition algorithm (DA) [1]. It was shown that in the presence of cyclic functional dependencies the procedure proposed in DA results in bad design (only a local XNF can be obtained, i.e. only subschemas of the schema can be transformed into XNF).

Acknowledgement: This work was supported in part by the Polish Ministry of Science and Higher Education under grant N N516 369536.

References

[1] M. Arenas and L. Libkin, "A normal form for XML documents." *ACM Trans. Database Syst.*, vol. 29, pp. 195–232, 2004.

[2] E. Codd, "Further normalization of the data base relational model," *R. Rustin (ed.): Database Systems, Prentice Hall, and IBM Research Report RJ 909*, pp. 33–64, 1972.

[3] E. F. Codd, "Recent investigations in relational data base systems," in *IFIP Congress*, 1974, pp. 1017–1021.

[4] S. Kolahi, "Dependency-Preserving Normalization of Relational and XML Data," *Database Programming Languages, DBPL 2005, Lecture Notes in Computer Science*, vol. 3774, pp. 247–261, 2005.

[5] ——, "Dependency-preserving normalization of relational and XML data," *Journal of Computer and System Sciences*, vol. 73, no. 4, pp. 636–647, 2007.

[6] S. Kolahi and L. Libkin, "On redundancy vs dependency preservation in normalization: an information-theoretic study of 3NF," in *PODS '06*. ACM, New York, USA, 2006, pp. 114–123.

[7] P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, and W. C. Tan, "Reasoning about keys for XML," *Information Systems*, vol. 28, no. 8, pp. 1037–1063, 2003.

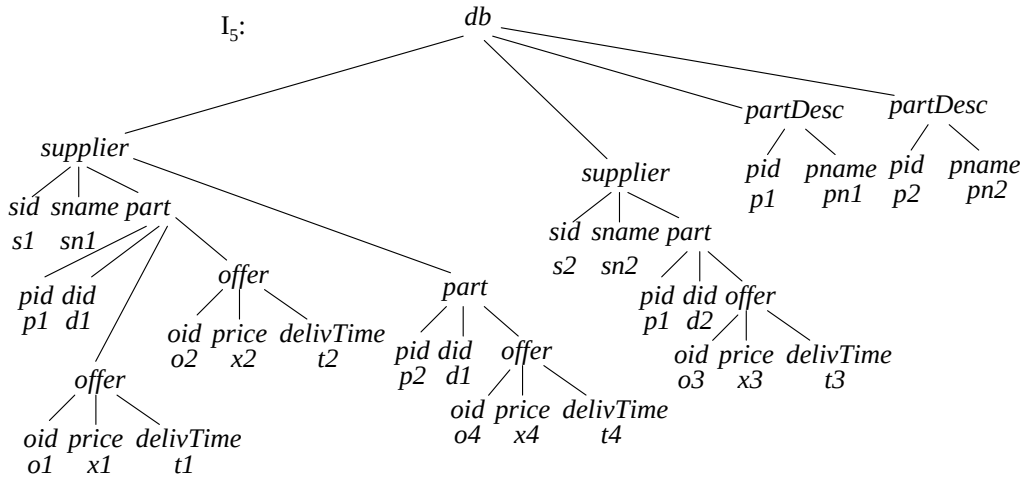

Figure 16: Instance of S_5 from Figure 15

[8] W. Fan and J. Siméon, "Integrity constraints for XML," *J. Comput. Syst. Sci.*, vol. 66, no. 1, pp. 254–291, 2003.

[9] M. W. Vincent, J. Liu, and M. K. Mohania, "On the equivalence between FDs in XML and FDs in relations," *Acta Inf.*, vol. 44, no. 3-4, pp. 207–247, 2007.

[10] M. W. Vincent and J. Liu, "Checking functional dependency satisfaction in XML," *Database and XML Technologies – XSym 2005, Lecture Notes in Computer Science*, vol. 3671, pp. 4–17, 2005.

[11] C. Yu and H. V. Jagadish, "XML schema refinement through redundancy detection and normalization," *VLDB Journal*, vol. 17, no. 2, pp. 203–223, 2008.

[12] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*. Redwood City: The Benjamin/Cummings, 1994.

[13] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Reading, Massachusetts: Addison-Wesley, 1995.

[14] M. Arenas and L. Libkin, "An information-theoretic approach to normal forms for relational and XML data," *J. ACM*, vol. 52, no. 2, pp. 246–283, 2005.

[15] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner, "Pathfinder: XQuery - the relational way," in *VLDB 2005*. ACM, 2005, pp. 1322–1325.

[16] S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, W. Yu, D. Tomic, A. Baras, B. Berg, D. Churin, and E. Kogan, "XQuery implementation in a relational database system," in *VLDB 2005*. ACM, 2005, pp. 1175–1186.

[17] P. Pigozzo and E. Quintarelli, "An algorithm for generating XML Schemas from ER Schemas," in *SEBD*, 2005, pp. 192–199.

[18] R. Schroeder and R. dos Santos Mello, "Improving query performance on XML documents: a workload-driven design approach," in *DocEng*. ACM, 2008, pp. 177–186.

[19] W. Xu and Z. M. Özsoyoglu, "Rewriting XPath queries using materialized views," in *VLDB 2005*, 2005, pp. 121–132.

[20] T. Pankowski, "XML data integration in SixP2P – a theoretical framework," in *EDBT Workshop Data Management in P2P Systems (DAMAP 2008),*ACM Digital Library, 2008, pp. 11–18.

[21] W. W. Armstrong, "Dependency structures of data base relationships," in *IFIP Congress*, 1974, pp. 580–583.

[22] P. A. Bernstein, "Synthesizing third normal form relations from functional dependencies," *ACM Transactions on Database Systtems*, vol. 1, no. 4, pp. 277–298, 1976.

[23] J. Rissanen, "Independent components of relations," *ACM Transactions on Database Systems*, vol. 2, no. 4, pp. 317–325, 1977.

[24] M. Arenas, "Normalization theory for XML," *SIGMOD Record*, vol. 35, no. 4, pp. 57–64, 2006.

[25] XML Schema Definition Language (XSDL) 1.1, 2007, www.w3.org/TR/xmlschema11-1.

[26] W. Martens, F. Neven, and T. Schwentick, "Simple off the shelf abstractions for XML schema," *SIGMOD Record*, vol. 36, no. 3, pp. 15–22, 2007.

[27] XML Path Language (XPath) 2.0, 2006, www.w3.org/TR/xpath20.

[28] P. P. Chen, "The entity-relationship model - toward a unified view of data," *ACM Transactions on Database Systtems*, vol. 1, no. 1, pp. 9–36, 1976.