Nesting-based Relational-to-XML Schema Translation

Dongwon Lee	Murali Mani	Frank Chiu	Wesley W. Chu
UCLA / CSD	UCLA / CSD	UCLA / CSD	UCLA / CSD
dongwon@cs.ucla.edu	mani@cs.ucla.edu	fchiu@ucla.edu	wwc@cs.ucla.edu

1 Introduction

XML is rapidly becoming one of the most widely adopted technologies for information exchange and representation on the World-Wide Web. With XML emerging as *the* data format of the Internet era, there is a substantial increase in the amount of data encoded in XML. However, the majority of everyday data is still stored and maintained in relational databases. Therefore, we expect the needs to convert such relational data into XML documents to grow substantially as well. In this paper, we study the problems in this conversion. Especially, we are interested in finding XML schema (e.g., DTD, XML-Schema, RELAX) that *best* describes the existing relational schema. Having the XML schema that precisely describes the semantics and structures of the original relational data is important to further maintain the converted XML documents in future. We first present a straightforward relational to XML translation algorithm, called *Flat Translation* (FT). Since FT maps the flat relational model to the flat XML model in a one-to-one manner, it does not utilize the regular expression of element content models at all. Then, we present our ongoing work, called *Nesting-based Translation* (NeT), to remedy the problems found in FT. NeT derives nested structures from a flat relational model by the use of the *nest* operator so that the resulting DTD is more intuitive and precise than otherwise.

Related Work: From XML to relational schema, several conversion algorithms have been proposed recently (e.g., [7, 5]). In this paper, on the contrary, we mainly concern conversion issues in the reverse direction (e.g., DB2XML, Oracle8i iFS). In DB2XML [8], an algorithm similar to our FT (thus shares similar problems) is introduced. In experimentation, we show NeT can generate a more precise DTD than DB2XML. In addition, there have been other DTD inference algorithms that take as "input" a set of XML documents [3] or a view description [6]. Unlike them, our approach takes a relational schema as input.

1.1 Input & Output Models

We first briefly define the input and output models for the translation. In relational databases, schema is typically created by SQL DDL (e.g., `CREATE`) statements. Therefore, by examining such DDL statements, one can find out the original schema information. Even if such DDL statements are not available, one can still infer the schema information by examining database tables – table and column names, key and foreign key information, etc. In this paper, regardless of how one acquired the schema information, we assume that the schema information is encoded in a vector $\mathbb{R}$ defined below.

First, we assume that the existence of a set $\widehat{T}$ of table names, a set $\widehat{C}$ of column names and a set $\widehat{b}$ of atomic base types (e.g., integer, char, string). When the name collision occurs, a column name $c \in \widehat{C}$ is qualified by a table name $t \in \widehat{T}$ using the "." notation (e.g., $t.c$).

Definition 1 (Relational Schema) A *relational schema* $\mathbb{R}$ is denoted by 4-tuple $\mathbb{R} = (T, C, P, \Delta)$, where: (1) T is a finite set of table names in $\widehat{T}$, (2) C is a function from a table name $t \in T$ to a set of column names $c \in \widehat{C}$, (3) P is a function from a column name c to its *column type definition*: i.e., $P(c) = \alpha$, where α is a 5-tuple (τ, u, n, d, f), where $\tau \in \widehat{b}$, u is either "unique" or "not_unique", n is either "nullable" or "not_nullable", d is a finite set of valid domain values of c or ϵ if not known, and f is a default value of c or ϵ if not known, and (4) Δ is a finite set of integrity constraints. For practical reasons, Δ includes only

the constraints that can be retrieved from relational databases via ODBC/JDBC connection such as key constraint ($\{c_i\} \stackrel{key}{\rightarrow} t$) and referential constraint ($c_1 \subseteq c_2$). □

Example 1. Consider two tables `student(`<u>`Sname`</u>`, Gender, `<u>`Advisor`</u>`, `<u>`Course`</u>`)` and `professor(`<u>`Pname`</u>`, Age)` where keys are underlined, and `Advisor` is a foreign key referencing `Pname` column. The column `Age` is integer type while the rest of the columns are string types. The column `Gender` can only be {M,F}. The column `Age` can be null. When student's advisor has not yet been decided, professor "J. Smith" will be the initial advisor. Student can have many advisors and take zero or more courses. Then, the relational schema can be denoted as $\mathbb{R}_1 = (T, C, P, \Delta)$, where

$$
\begin{array}{rcll}
T & = & \{student, professor\} & \\
C(student) & = & \{Sname, Gender, Advisor, Course\} & C(professor) = \{Pname, Age\} \\
P(Gender) & = & (string, not\_unique, nullable, \{M, F\}, \epsilon) & P(Sname) = (string, unique, not\_nullable, \epsilon, \epsilon) \\
P(Advisor) & = & (string, not\_unique, not\_nullable, \epsilon, \text{“}J.Smith''\text{)} & P(Course) = (string, not\_unique, nullable, \epsilon, \epsilon) \\
P(Pname) & = & (string, unique, not\_nullable, \epsilon, \epsilon) & P(Age) = (integer, not\_unique, nullable, \epsilon, \epsilon) \\
\Delta & = & \{\{Sname, Advisor, Course\} \stackrel{key}{\rightarrow} student, Pname \stackrel{key}{\rightarrow} professor, Advisor \subseteq Pname\} &
\end{array}
$$

Next, let us define the output model. Among a dozen XML schema languages recently proposed (e.g., DTD, XML-Schema, RELAX, XDR), in this paper, we focus on DTD due to its simplicity and popularity. Starting from the notations in [1], we define the DTD schema below. We assume that the existence of a set $\widehat{E}$ of element names, a set $\widehat{A}$ of attribute names and a set $\widehat{S}$ of string values. When needed, an attribute name $a \in \widehat{A}$ is qualified by the element names using the *path expression* notation (e.g., $e_1.e_2 \cdots e_n.a$, where $e_i \in \widehat{E}, 1 \leq i \leq n$). τ denotes a data type for an attribute in DTD: $\tau ::= S \mid ID \mid IDREF \mid IDREFS$, where S denotes a `string` type.

Definition 2 (DTD Schema) A *DTD schema* $\mathbb{D}$ is denoted by 6-tuple $\mathbb{D} = (E, A, M, P, r, \Sigma)$, where: (1) E is a finite set of element names in $\widehat{E}$, (2) A is a function from an element name $e \in E$ to a set of attribute names $a \in \widehat{A}$, (3) M is a function from an element name $e \in E$ to its *element type definition*: i.e., $M(e) = \alpha$, where α is a regular expression: $\alpha ::= S \mid e' \mid \epsilon \mid \alpha + \alpha \mid \alpha, \alpha \mid \alpha^? \mid \alpha^* \mid \alpha^+$, where $S \in \widehat{S}$, $e' \in E$, ϵ denotes the empty element, "+" for the union, "," for the concatenation, "$\alpha^?$" for zero or one occurrence, "α^*" for the Kleene closure, and "α^+" for "α, α^*", (4) P is a function from an attribute name a to its *attribute type definition*: i.e., $P(a) = \beta$, where β is a 4-tuple (ϕ, n, d, f), where $\phi \in \tau$, n is either "nullable" or "not_nullable", d is a finite set of valid domain values of a or ϵ if not known, and f is a default value of a or ϵ if not known, and (5) r is a finite set of root element name $e \in E$; Σ is a finite set of integrity constraints that include only key and referential constraints. □

Example 2. An element `book` with attributes `<!ATTLIST book ISBN ID #REQUIRED Title CDATA 'Untitled' Authors IDREFS #IMPLIED>` can be encoded as $\mathbb{D}_2 = (E, A, M, P, r, \Sigma)$, where

$$
\begin{array}{rcllrcl}
E & = & \{book\} & & A(book) & = & \{ISBN, Title, Authors\} \\
M(book) & = & \epsilon & & P(ISBN) & = & (ID, not\_nullable, \epsilon, \epsilon) \\
P(Title) & = & (S, nullable, \epsilon, \text{“}Untitled''\text{)} & & P(Authors) & = & (IDREFS, nullable, \epsilon, \epsilon) \\
r & = & \{book\} & & \Sigma & = & \{ISBN \stackrel{key}{\rightarrow} book\}
\end{array}
$$

2 Translation from $\mathbb{R}$ to $\mathbb{D}$

XML model uses two basic building blocks to construct XML documents – attribute and element. A few basic limitations inherited from XML model include: (1) attribute does not have order semantics while element has, (2) both support a string type, and (3) element can express multiple occurrence better than attribute. The detailed capabilities of those, however, vary depending on the chosen XML schema language. In translating $\mathbb{R}$ to $\mathbb{D}$, therefore, one can either use attribute or element in $\mathbb{D}$ to represent the same entity in $\mathbb{R}$ (e.g, a column with string type in $\mathbb{R}$ can be translated to either attribute with `CDATA` type or element with `PCDATA` type in $\mathbb{D}$).

To increase the flexibility of the algorithms, we assume that there are two modes – "attribute-oriented" and "element-oriented". Depending on the mode, an algorithm can selectively translate an entity in $\mathbb{R}$ to either attribute or element if both can capture the entity correctly. However, if the chosen XML schema language requires attribute or element for an entity (e.g., a key column in $\mathbb{R}$ needs to be translated to an

attribute with `ID` type in $\mathbb{D}$), we assume that algorithm follows the limitations. In addition, in this paper, we focus on the separate conversion of the "single" table. Current naive algorithm to support the conversion of "multiple" inter-connected tables has the following two simple heuristics: (1) convert each table separately and glue them with concatenation operator (i.e., ",") in the element definition, and (2) foreign key constraint is handled by including the referencing element into the element type definition of the referenced element. More sophisticated algorithms are currently under development.

2.1 Flat Translation

The simplest translation method is to translate (1) tables in $\mathbb{R}$ to elements in $\mathbb{D}$ and (2) columns in $\mathbb{R}$ to attributes (in attribute-oriented mode) or elements (in element-oriented mode) in $\mathbb{D}$. These two modes are analogous except that element-oriented mode adds unnecessary order semantics to the resulting schema; note that relational model has orderless set semantics. Since $\mathbb{D}$ represents the "flat" relational tuples faithfully, this method is called *Flat Translation (FT)*. The general procedure of the **Flat Translation** is straightforward and omitted due to space constraints.

FT is a simple and effective translation algorithm, but it has some problems. As the name implies, FT translates the "flat" relational model to "flat" XML model in one-to-one manner. Thus, for every tuple in $\mathbb{R}$, one corresponding element must be generated. Therefore, the "non-flat" features in XML model through regular expressions (e.g., "*", "+") are not being utilized at all. Consider the case of `paper` where it can have a single `title`, but multiple `authors` and `keywords`. In relational model, this can be modeled as `paper(title, author, keyword)`. Suppose this is translated to (1) `<!ELEMENT paper (title, author, keyword)>` and (2) `<!ELEMENT paper (title, author+, keyword+)>`. In general, from the users' perspective, the second DTD is easier to understand than the first DTD, since it represents the real world more accurately: the first DTD implies that `paper` can have only one `author` and `keyword`. The main reason of this discrepancy between the users' perception and the representation is that in relational model, concepts need to be *flattened out* to fit into the model. Therefore, since XML model allows hierarchical nesting, it would be desirable to *unflatten* concepts to make such structures if preferred so.

2.2 Nesting-based Translation

To remedy the problems of FT, one needs to utilize various *element content models* of XML. Towards this goal, we propose to use the *nest* operator [4]. Our idea is to find a best element content model that uses the α^* or α^+ using the *nest* operator. First, let us define the *nest* operator. Informally, for a table t with a set of columns C, *nesting* on a non-empty column $X \in C$ collects all tuples that agree on the remaining columns $C - X$ into a set[1]. Formally,

Definition 3 (Nest) [4]. Let t be a n-ary table with column set C. Let further $X \in C$ and $\overline{X} = C - X$. For each $(n-1)$-tuple $\gamma \in \Pi_{\overline{X}}(t)$, we define an n-tuple γ^* as follows:

$$\left.\begin{array}{lcl} \gamma^*[\overline{X}] & = & \gamma \\ \gamma^*[X] & = & \{\kappa[X] \mid \kappa \in t \wedge \kappa[\overline{X}] = \gamma\} \end{array}\right\} \quad \textit{then,} \quad nest_X(t) = \{\gamma^* \mid \gamma \in \Pi_{\overline{X}}(t)\}$$ □

After $nest_X(t)$, if the column X has only a set with "single" value $\{v\}$, then we say that *nesting failed* and treat $\{v\}$ and v interchangeably (i.e., $\{v\} = v$). Thus when nesting failed, the following is true: $nest_X(t) = t$. Otherwise, if the column X has a set with "multiple" values $\{v_1, ..., v_k\}$ with $k \geq 2$, we say that *nesting succeeded*.

Example 3. Consider a table R in Table 1. In computing $nest_A(R)$ at (b), the first, third, and fourth tuples of R agree on their values in columns (B, C) as (a, 10), while their values of the column A are all different. Therefore, these different values are grouped (i.e., nested) into a set {1,2,3}. The result is the first tuple of the table $nest_A(R)$ – ({1,2,3}, a, 10). Similarly, since the sixth and seventh tuples of R agree on their values as (b, 20), they are grouped to a set {4,5}. In computing $nest_B(R)$ at (c), there are no tuples in R that agree on the values of the columns (A, C). Therefore, $nest_B(R) = R$. In computing $nest_C(R)$ at (d), since the first two tuples of R – (1, a, 10) and (1, a, 20) – agree on the values of the columns (A, B), they are grouped to (1, a, {10,20}). Nested tables (e) through (j) are constructed similarly.

[1] Here, we only consider single attribute nesting.

A	B	C
1	a	10
1	a	20
2	a	10
3	a	10
4	b	10
4	b	20
5	b	20

(a) R

A+	B	C
{1,2,3}	a	10
1	a	20
4	b	10
{4,5}	b	20

(b) $nest_A(R)$

A	B	C
1	a	10
1	a	20
2	a	10
3	a	10
4	b	10
4	b	20
5	b	20

(c) $nest_B(R) = R$

A	B	C+
1	a	{10,20}
2	a	10
3	a	10
4	b	{10,20}
5	b	20

(d) $nest_C(R)$

A+	B	C
{1,2,3}	a	10
1	a	20
4	b	10
{4,5}	b	20

(e) $nest_B(nest_A(R)) = nest_C(nest_A(R))$

A+	B	C+
1	a	{10,20}
{2,3}	a	10
4	b	{10,20}
5	b	20

(f) $nest_A(nest_C(R))$

A	B	C+
1	a	{10,20}
2	a	10
3	a	10
4	b	{10,20}
5	b	20

(g) $nest_B(nest_C(R))$

A+	B	C
{1,2,3}	a	10
1	a	20
4	b	10
{4,5}	b	20

(h) $nest_C(nest_B(nest_A(R))) = nest_B(nest_C(nest_A(R)))$

A+	B	C+
1	a	{10,20}
{2,3}	a	10
4	b	{10,20}
5	b	20

(i) $nest_B(nest_A(nest_C(R))) = nest_A(nest_B(nest_C(R)))$

Table 1: A relational table R and its various nested forms. Column names containing a set after nesting (i.e., nesting succeeded) are appended by "+" symbol.

Our idea of using the *nest* operator is as follows: If nesting on column X succeeded, then the column X can be *unflattened* to contain multiple occurrences of values so that it can capture a more realistic model. That is, we use the element content model of either $*$ (if X is nullable) or $+$ (if X is not nullable) if the nesting on the column X succeeded. For instance, consider the nested tables in Table 1. Assuming columns A, B, C are not nullable, if the nesting succeeded, the corresponding column names are appended by "+" symbol.

Since the *nest* operator requires scanning of the entire tuples in a given table, it can be quite expensive. In addition, as shown in Example 3, there are various ways to nest the given table. Therefore, it is important to find the best element content model with the least number of nesting.

Lemma 1. *Applying the nest operator on a non-prime column*[2] X *yields no changes.* ∎

PROOF. Consider a table t with columns C. If the column X is not part of key columns Z, then the remaining columns $\overline{X}$ must contain Z (i.e., $\overline{X} \supseteq Z$). Since the values of Z are unique by the definition of the "key", the values of Z's superset, $\overline{X}$, must be also unique (i.e., Z is a candidate key while $\overline{X}$ is a super key). Thus, no tuples can agree on the values of $\overline{X}$. By the definition of the *nest* operator, therefore, nesting on X would yield the same table as before. (q.e.d)

COROLLARY. For any nested table $nest_X(t)$, $\overline{X} \rightarrow X$ holds. (q.e.d)

The Corollary states that after applying the nesting operation of column X, the remaining columns $\overline{X}$ becomes a super key. Fischer et al. [2] have proved that functional dependencies are preserved against nesting as follows:

Lemma 2. [2] *If X, Y, Z are columns of t, then: $t : X \rightarrow Y \Longrightarrow nest_Z(t) : X \rightarrow Y$* ∎

Now, using Lemmas 1 and 2, we arrive at the following useful property:

Lemma 3. *Using the* falling factorial power *notation "x to the m falling" as $x^{\underline{m}}$, for a table t with n columns where m ($m \leq n$) columns constitute the key, the maximum number T of the necessary nesting is: $T = \sum_{k=1}^{m} m^{\underline{k}}$* ∎

PROOF. Let us denote the n columns as C and m key columns as K (i.e., $K \subset C$). By the definition of the key, $t : K \rightarrow \overline{K}$. By Lemma 1, we first only need to nest along m columns in K, excluding the remaining $n - m$ columns. Suppose we pick the column $X \in K$ as the first one to nest. After nesting,

[2] A *prime column* is a column which participates in at least one key.

by Corollary and Lemma 2, we have $nest_X(t)$:$\{K \rightarrow \overline{K},\ \overline{X} \rightarrow X\}$. In choosing a second column to nest against the $nest_X(t)$, we need to select a key column Y that belongs to the left hand side of both of the above constraints. That is, $Y \in K \cap \overline{X} = K - X$. Since K has m columns, $K - X$ has $m - 1$ columns. Therefore, in choosing a second column, we have $m-1$ columns to nest. Continuing this, we have: $m + m(m-1) + \cdots + m(m-1)\ldots(2)(1) \quad = \quad \sum_{k=1}^{m} m^{\underline{k}}$ (q.e.d)

Example 4. Consider a table R in Table 1 again. Now, suppose attributes A and C constitute a key for R. Since nesting on the same column repeatedly is not useful [4], there is no need to construct, for instance, $nest_A(nest_A(R))$. Since nesting on a non-key column is not useful by Lemma 1, nesting along the column B (e.g., $nest_B(R)$ at (c)) can be avoided. Furthermore, the functional dependency (i.e., $AC \stackrel{key}{\rightarrow} R = AC \rightarrow \overline{AC} = AC \rightarrow B$) persists after nesting on either the column A or C by Lemma 2. Hence, by Lemma 1 again, nesting on the column B is still not useful and can be avoided (e.g., $nest_B(nest_A(R))$ at (e), $nest_B(nest_C(R))$ at (g), $nest_B(nest_C(nest_A(R)))$ at (h), $nest_B(nest_A(nest_C(R)))$ at (i)). Consequently, one needs to construct only the following nested tables: $nest_A(R)$ at (b), $nest_C(R)$ at (d), $nest_C(nest_A(R))$ at (e), $nest_A(nest_C(R))$ at (f).

As we have shown, when the key information is available, the cost for *nest* operator can be minimized. However, when such information is not known or when the given tables are intermediate ones generated by queries, *nest* operation must be applied for all possible combinations. After applying the *nest* operator to the given table repeatedly, there can be still several nested tables where the nesting succeeded. In general, the choice of the final schema should take into consideration of the semantics and usages of the underlying data or application. In this paper, for the sake of simplicity, we choose as the final schema the nested table where the most number of nesting succeeded. The general procedure of the **Nesting-based Translation (NeT)** is as follows:

1. Each table t_i in $\mathbb{R}$ is translated to an element e_i in $\mathbb{D}$: $E = \bigcup_{\forall i}\{e_i\}$.
2. For each table t_i in $\mathbb{R}$, let us denote the final nested table as $t_i(c_1, ..., c_{k-1}, c_k, ..., c_n)$, where nesting succeeded on the columns $\{c_1, ..., c_{k-1}\}$. If $k = 1$ (i.e., no nesting succeeded), follow the flat translation. Otherwise, do the following:
   (a) For each column c_i $(1 \leq i \leq k-1)$, if c_i was defined as "nullable" in P of $\mathbb{R}$, then the content model is $M(e_i) = (c_i^*, ...)$, otherwise $M(e_i) = (c_i^+, ...)$
   (b) For each column c_j $(k \leq j \leq n)$,
   - (element-oriented mode) if c_j was defined as "nullable" in P of $\mathbb{R}$, the content model is $M(e_i) = (..., c_j^?)$, otherwise $M(e_i) = (..., c_j)$.
   - (attribute-oriented mode) if c_j is translated to a_j, then $A(e_i) = \bigcup_{\forall j}\{a_j\}$ and if c_j was defined as "nullable" in P of $\mathbb{R}$, $P(a_j) = (S,$ nullable, $d,\ f)$, otherwise $P(a_j) = (S,$ not_nullable, $d,\ f)$.
3. All elements e_i in $\mathbb{D}$ become roots: $r = \bigcup_{\forall i}\{e_i\}$; Copy Δ in $\mathbb{R}$ into Σ in $\mathbb{D}$.

3 Experimental Results

In this section, we compare the preliminary results of FT and NeT with that of DB2XML v 1.3 [8]. Consider, for instance, the `Orders` table (containing 830 tuples) found in MS Access NorthWind sample database.

```
Orders (CustomerID, EmployeeID, ShipVia, ShipAddress, ShipCity, ShipCountry, ShipPostalCode)
```

Table 2 shows the DTDs generated by DB2XML, FT in attribute-oriented mode, NeT in both element-oriented mode and attribute-oriented mode, respectively.

In (a), DB2XML always uses element to represent columns of a table. To represent whether the column is nullable or not, DB2XML adds a special attribute `ISNULL` to every element: i.e., "`ISNULL = true`" means the column is nullable. In (b), FT in attribute-oriented mode uses `#IMPLIED` to represent whether the column is nullable or not. Observe that both DB2XML and FT share the same problem of translating "flat" relational

```
<!ELEMENT Orders (CustomerID,EmployeeID,ShipVia,ShipAddress,
                  ShipCity,ShipCountry,ShipPostalCode)>
<!ELEMENT CustomerID      (#PCDATA)>
<!ATTLIST CustomerID      ISNULL (true|false) #IMPLIED>
<!ELEMENT EmployeeID      (#PCDATA)>
<!ATTLIST EmployeeID      ISNULL (true|false) #IMPLIED>
<!ELEMENT ShipVia         (#PCDATA)>
<!ATTLIST ShipVia         ISNULL (true|false) #IMPLIED>
...
<!ELEMENT ShipCountry     (#PCDATA)>
<!ATTLIST ShipCountry     ISNULL (true|false) #IMPLIED>
<!ELEMENT ShipPostalCode  (#PCDATA)>
<!ATTLIST ShipPostalCode  ISNULL (true|false) #IMPLIED>
```

(a) DB2XML

```
<!ELEMENT  Orders  (EMPTY)>
<!ATTLIST  Orders
           CustomerID     CDATA #IMPLIED
           EmployeeID     CDATA #IMPLIED
           ShipVia        CDATA #IMPLIED
           ShipAddress    CDATA #IMPLIED
           ShipCity       CDATA #IMPLIED
           ShipCountry    CDATA #IMPLIED
           ShipPostalCode CDATA #IMPLIED>
```

(b) FT in attribute-oriented mode

```
<!ELEMENT  Orders  (CustomerID,EmployeeID+,ShipVia*,ShipAddress?,
                    ShipCity?,ShipCountry?,ShipPostalCode?)>
<!ELEMENT  CustomerID     (#PCDATA)>
<!ELEMENT  EmployeeID     (#PCDATA)>
<!ELEMENT  ShipVia        (#PCDATA)>
<!ELEMENT  ShipAddress    (#PCDATA)>
<!ELEMENT  ShipCity       (#PCDATA)>
<!ELEMENT  ShipCountry    (#PCDATA)>
<!ELEMENT  ShipPostalCode (#PCDATA)>
```

(c) NeT in element-oriented mode

```
<!ELEMENT  Orders  (EmployeeID+,ShipVia*)>
<!ATTLIST  Orders
           CustomerID      CDATA #REQUIRED
           ShipAddress     CDATA #IMPLIED
           ShipCity        CDATA #IMPLIED
           ShipCountry     CDATA #IMPLIED
           ShipPostalCode  CDATA #IMPLIED>
<!ELEMENT  EmployeeID  (#PCDATA)>
<!ELEMENT  ShipVia     (#PCDATA)>
```

(d) NeT in attribute-oriented mode

Table 2: DTDs generated by different algorithms.

schema to "flat" XML schema. In both (c) and (d), NeT finds two columns `EmployeeID` amd `ShipVia` can be nested. Intuitively, the new schema infers that for each `CustomerID`, multiple non-zero `EmployeeID` and multiple `ShipVia` can exist. Also NeT finds that `CustomerID` is mandatory column. To ensure this property, (c) adds no suffix such as `?` or `*` to `CustomerID` sub-element and (d) uses `#REQUIRED` construct explicitly. Not only DTDs found by NeT are more succinct than one by DB2XML, but we also found that they are more accurate and intuitive.

4 Conclusion

We have presented two relational-to-XML conversion algorithms. Especially, NeT algorithm is capable of generating a more precise and intuitive XML schema from relational inputs using the *nest* operator developed in nested relational model. We show a proof of concept by comparing (and showing the superiority of) the results of our implementation and that of DB2XML tool. Extension of the current work for schema with multiple (inter-connected) tables is on-going.

References

[1] W. Fan and J. Siméon. "Integrity Constraints for XML". In *ACM PODS*, Dallas, TX, May 2000.

[2] P. C. Fischer, L. V. Saxton, S. J. Thomas, and D. V. Gucht. "Interactions between Dependencies and Nested Relational Structures". *J. Computer and System Sciences (JCSS)*, 31(3):343–354, Dec. 1985.

[3] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. "XTRACT: A System for Extracting Document Type Descriptors from XML Documents". In *ACM SIGMOD*, Dallas, TX, May 2000.

[4] G. Jaeschke and H.-J. Schek. "Remarks on the Algebra of Non First Normal Form Relations". In *ACM PODS*, Los Angeles, CA, Mar. 1982.

[5] D. Lee and W. W. Chu. "Constraints-preserving Transformation from XML Document Type Definition to Relational Schema". In *Int'l Conf. on Conceptual Modeling (ER)*, Salt Lake City, UT, Oct. 2000.

[6] Y. Papakonstantinou and P. Velikhov. "Enhancing Semistructured Data Mediators with Document Type Definitions". In *IEEE ICDE*, Sydney, Austrialia, Mar. 1999.

[7] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. "Relational Databases for Querying XML Documents: Limitations and Opportunities". In *VLDB*, Edinburgh, Scotland, Sep. 1999.

[8] V. Turau. "Making Legacy Data Accessible for XML Applications", 1999. http://www.informatik.fh-wiesbaden.de/~turau/veroeff.html.