XML to Relational Conversion using Theory of Regular Tree Grammars

Murali Mani*
UCLA / CSD
mani@cs.ucla.edu

Dongwon Lee
UCLA / CSD
dongwon@cs.ucla.edu

Abstract

In this paper, we study the different steps of translation from XML to relational models, while maintaining semantic constraints. Our work is based on the theory of regular tree grammars, which provides a useful formal framework for understanding various aspects of XML schema languages. We first study two normal form representations for regular tree grammars. The first normal form representation, called NF1, is used in the two scenarios: (a) Several document validation algorithms use the NF1 representation as the first step in the validation process for efficiency reasons, and (b) NF1 representation can be used to check whether a given schema satisfies the structural constraints imposed by the schema language. The second normal form representation, called NF2, forms the basis for conversion of a set of type definitions in a schema language L_1 that supports union types (e.g., XML-Schema), to a schema language L_2 that does not support union types (e.g., SQL), and is used as the first step in our XML to relational conversion algorithm.

1 Introduction

The theory of regular tree grammars [16, 6] provides an excellent framework for understanding various aspects of XML schema[1] languages [14], and have been actively used in many applications including XML document processing (e.g., XQuery from W3C [4] and XDuce [9]) and XML document validation algorithms (e.g., RELAX, TREX, and RELAX-NG [5]). It is also used for analyzing the expressive power and closure properties of the different schema language proposals [14].

Our emphasis, in this paper, is on understanding the data modeling aspects of XML schemas. XML schema provides several unique data modeling features, e.g., union types, that are not present in traditional database models such as the relational model, and this makes this study challenging. Several problems have been studied regarding the data modeling aspects of XML schemas. The foundations of the current work are based on our earlier work [13]. Here we describe how entities and relationships, which form the basis of data modeling, can be represented using the features provided by XML, such as elements, attributes, parent-child relationships, ID-IDREF attributes, and using inclusion dependencies. We further describe how the various definitions in a given XML schema can be mapped to entities and relationships. For example, a parent-child relationship from book to author represented as book → author* in XML can be mapped to an ordered 1:many relationship, where for every book, we have an ordered list of authors.

A related area that has generated a lot of interests is mapping from relational to XML models [11, 12]. In [12], we represent the relationships expressed using inclusion dependencies in the relational model as parent-child relationships, as ID-IDREF attributes, and sometimes as inclusion dependencies in the XML model. For example, if we have a relation book and a relation author, and author has a foreign key referencing book, then we can represent it in XML as book → author*. In this paper, we focus on the reverse conversion from XML to relational schemas. This is necessary for storing XML documents using a relational database, and has been widely studied [7, 15, 8, 2, 10].

*This author is partially supported by NSF grants 0086116, 0085773, 9817773.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

[1] We differentiate two terms – XML schema(s) and XML-Schema. The former is a general term for schema in XML model while the latter refers in particular to the XML schema language proposed by W3C [17].

Our approach is different from the existing approaches in that we consider semantic constraints in the XML model, and they are captured in our resulting relational schema. Further, our conversion is based on the strong and clean mathematical foundations provided by regular tree grammars.

In this paper, we first give a theoretical exposition for XML schemas using regular tree grammar theory, and provide two normal form representations – NF1 and NF2. The first step in several document validation algorithms based on tree automata [14] is to represent the given XML schema in NF1 to give better performance. Also the NF1 representation is used for checking the validity of a given schema against the constraints imposed by the schema language. For example, it is used to check whether a given tree grammar is a single type tree grammar [14].

An important application of NF2 presented in this paper is in the conversion from XML to relational models. However, the usefulness of NF2 is more general – NF2 provides the theoretical basis for mapping the type definitions in a language that supports union types, such as XML schemas, to a language that does not support union types, such as SQL.

After we describe the two normal form representations of regular tree grammars, we describe algorithms to map an XML schema, starting from the NF2 representation, to a relational schema. There are several issues in this mapping that we study:

- Given that the relational schema cannot express all the constraints in the XML schema, what is the useful and meaningful subset of constraints that should be mapped?. Answering this question gives our schema simplification step, where the complex content model is converted into a simpler content model that can be represented in a relational schema.
- Inlining [15] is a technique that is used for generating more meaningful as well as "efficient" relational schemas. We describe the inlining algorithm which can be used for any general regular tree grammar.
- How do we handle collection types, recursion, and IDREF and IDREFS attributes defined in XML schema?
- How do we maintain semantic constraints such as key constraints and inclusion dependencies?

1.1 Related Work

Regular tree grammars and automata have been used by the authors in [14] to compare the expressive power and closure properties of the different XML schema language proposals. Furthermore, they are used for several document validation algorithms as presented in [14], and in several products such as the RELAX NG [5] validator, and XDuce [9] validator. The NF1 representation presented in this paper provides a useful addition to existing tools for document validation as well as schema validation.

Several techniques have been provided in the past for mapping an XML schema to a relational schema. In [7], the authors apply data mining techniques on semistructured data to find frequent patterns using a combination of the data instance as well as the query mix. The frequent patterns are stored in "optimized" relational storage, and the remaining patterns are stored in an overflow storage. This technique has the drawback that it requires integration of the relational and the overflow systems. In [8], the authors consider several mapping techniques, where an edge in the document $X\ e\ Y$ (i.e., Y is the value of the attribute of X called e) is mapped as three columns, X, e, Y. The drawback with this approach is the difficulty in maintaining semantic constraints. For example, suppose the value of the attribute e is a key for X, then it is difficult to specify this key constraint in the output relational schema.

The techniques presented in [15, 2] are closer to our goals of maintaining semantic constraints. We borrow some of our steps from them – our recursion elimination step is borrowed from [15], and the iterative improvement of the relational schema based on data and query statistics is borrowed from [2]. However, there are significant differences between our approaches. Maintaining semantic constraints was not the focus of both the above techniques, and hence they cannot capture several semantic constraints.

The techniques provided in [10] try to capture the semantic constraints in the relational schema. However they attempt to capture mainly cardinality constraints using equality generating dependencies, and tuple generating dependencies. Our work focuses on relationships and key constraints and maintains them in the relational schema.

1.2 Roadmap

In Section 2, we first define regular tree grammars. In Section 3, we define NF1 representation for a regular tree grammar, and describe how a regular tree grammar can be converted to NF1. Also, we present two applications of the NF1 representation. In Section 4, we define NF2 representation for a regular tree grammar, and describe how we can obtain an NF2 representation for a given regular tree grammar. In Section 5, we describe the different steps in mapping a given XML schema to relational schema. Finally, concluding remarks and future directions are discussed in Section 6.

2 Regular Tree Grammar

The structural specification of an XML schema is a regular tree grammar. We define a regular tree grammar below, borrowed from [14].

Definition 1 (Regular Tree Grammar) *A regular tree grammar (RTG) is a 4-tuple $G = (N, T, S, P)$, where:*

- *N is a finite set of* non-terminals,
- *T is a finite set of* terminals,
- *S is a set of* start symbols, *where S is a subset of N,*
- *P is a finite set of production rules of the form $X \rightarrow a\ RE$, where $X \in N$, $a \in T$, and RE is a regular expression over N; X is the left-hand side, a RE is the right-hand side, and RE is the* **content model** *of this production rule.* □

Example 1. The grammar $G_1 = (N, T, S, P)$ given in Table 1 is a regular tree grammar. For instance, $Author1 \rightarrow author(Son^*)$ is a production rule, whose left-hand side, right-hand side and the content model are $Author1$, $author(Son^*)$, and (Son^*), respectively. □

	N	=	$\{Book, Author1, Author2, Pub, Library, Museum, Son, Daughter\}$
	T	=	$\{book, author, publisher, library, museum, son, daughter\}$
	S	=	$\{Book\}$
P:	$Book$	$\rightarrow$	$book(Author1^*, Pub, (Library + Museum))$
	$Book$	$\rightarrow$	$book(Author2^*, Pub, Library)$
	$Author1$	$\rightarrow$	$author(Son^*)$
	$Author2$	$\rightarrow$	$author(Daughter^*)$
	Pub	$\rightarrow$	$publisher(\epsilon)$
	$Library$	$\rightarrow$	$library(\epsilon)$
	$Museum$	$\rightarrow$	$museum(\epsilon)$
	Son	$\rightarrow$	$son(\epsilon)$
	$Daughter$	$\rightarrow$	$daughter(\epsilon)$

Table 1: An example regular tree grammar G_1

A production rule $X \rightarrow a\ RE$ means that the non-terminal X can generate a tree with a as the root, and with children that "match" RE. In G_1, $Author1$ generates a tree with *author* as the root, and children that match Son^*. The set of trees that can be generated from any start symbol forms the language generated by the given regular tree grammar. For instance, a tree generated by G_1 is shown in Figure 1.

3 First Normal Form (NF1) for Regular Tree Grammars

In this section, we introduce the NF1 representation for regular tree grammars. NF1 representation requires

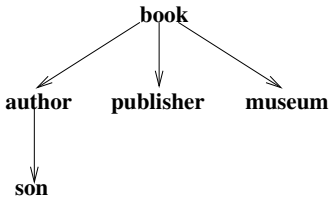

Figure 1: An example tree that is generated by G_1

that for every non-terminal, there must be *at most* one rule that produces a tree with a particular terminal as the root.

Definition 2 (NF1) *A regular tree grammar is said to be in* NF1 *if no two production rules have the same non-terminal in the left-hand side and the same terminal in the right hand side.* □

In other words, a regular tree grammar in NF1 does not have two rules of the form $X \rightarrow a\ RE_1$, and $X \rightarrow a\ RE_2$.

Example 2. The regular tree grammar G_1 in Example 1 is not in NF1. There are two rules, $Book \rightarrow book(Author1^*, Pub, (Library + Museum))$, and $Book \rightarrow book(Author2^*, Pub, Library)$, which have the same non-terminal in the left-hand side and the same terminal in the right hand side. □

Converting a given regular tree grammar to NF1 is straightforward: for every two rules of the form $X \rightarrow a\ RE_1$ and $X \rightarrow a\ RE_2$, replace the two rules by one rule as $X \rightarrow a\ (RE_1 + RE_2)$. This algorithm is outlined in Table 2.

Convert a regular tree grammar $G = (N, T, S, P)$ to NF1.

1. If there does not exist two rules in G of the form $X \rightarrow a\ RE_1$, and $X \rightarrow a\ RE_2$, return G.
2. Else replace the two rules with one rule $X \rightarrow a\ (RE_1 + RE_2)$, and go to step 1.

Table 2: RTG to NF1 algorithm.

Example 3. Converting G_1 to NF1, we obtain the regular tree grammar G_3 in Table 3.

The NF1 representation of regular tree grammars is used in at least two different XML application scenarios. One application scenario is in document validation, where a given document is verified whether it is valid against a given XML schema. Different document validation algorithms are discussed in [14], which form the basis of several implementations. Many of these algorithms convert a given regular tree grammar into its NF1 representation as the first step for efficiency reasons. For example, consider the document validation algorithm based on non-deterministic bottom-up tree automata [14]. First, the given regular

	N	$=$	$\{Book, Author1, Author2, Pub,$ $Library, Museum, Son, Daughter\}$
	T	$=$	$\{book, author, publisher,$ $library, museum, son, daughter\}$
	S	$=$	$\{Book\}$
$P:$	$Book$	$\rightarrow$	$book((Author1^*, Pub,$ $(Library + Museum)) +$ $(Author2^*, Pub, Library))$
	$Author1$	$\rightarrow$	$author(Son^*)$
	$Author2$	$\rightarrow$	$author(Daughter^*)$
	Pub	$\rightarrow$	$publisher(\epsilon)$
	$Library$	$\rightarrow$	$library(\epsilon)$
	$Museum$	$\rightarrow$	$museum(\epsilon)$
	Son	$\rightarrow$	$son(\epsilon)$
	$Daughter$	$\rightarrow$	$daughter(\epsilon)$

Table 3: G_3 = NF1 representation for G_1

tree grammar is converted to NF1. During validation when a start tag is encountered, we identify those production rules $X \rightarrow a\ RE$, such that a is the terminal of this tag. There may be multiple production rules with a on the right hand side, but all these rules will have different non-terminals as the grammar is in NF1. When the corresponding end-tag is encountered, we check if the non-terminals assigned to the children belong to the language generated by RE. If so, X is one of the valid non-terminals for this element. Otherwise, X is not a valid non-terminal for this element. The advantage of the NF1 representation is that we need to check the validity of the non-terminals assigned to the children against fewer regular expressions, which would yield better performance. For example, the children of *book* have to be checked against two regular expressions in G_1, as opposed to one regular expression in G_3.

Another application of NF1 is in schema validation, where a given schema is verified whether it satisfies the constraints imposed by the schema language. Different schema language proposals impose different constraints on the possible set of XML schemas that are valid with respect to that schema language. For example, DTD forms local tree grammars, and XML-Schema forms single type tree grammars [14]. Now suppose given a schema, we have to check whether it is valid with respect to that language. For example, consider verifying whether G_1 is a valid schema in XML-Schema or not. So we have to check whether G_1 is a single type tree grammar. Single type tree grammars place the restriction that there *should* not exist two different non-terminals that are start symbols, or that occur in the content model of the same non-terminal that "compete" with each other[14]. Two non-terminals are said to compete with each other if they have production rules that generate trees with the same terminal as the root. To check if G_1 is a single type tree grammar, we first convert G_1 to its NF1 representation, G_3. Now we check whether there exists a production rule, where the non-terminals in its content model compete with each other. We find that non-terminals $Author1$ and $Author2$ compete with each other and occur in the same production rule. Therefore G_1 is not a valid schema if XML-Schema is used as the schema language.

4 Second Normal Form (NF2) for Regular Tree Grammars

In this section, we define the NF2 representation for regular tree grammars. NF2 forms the basis for conversion of type definitions in a programming language L_1 that supports union types (e.g., XML-Schema), to a programming language L_2 that does not support union types (e.g., SQL).

Definition 3 (NF2) *A regular tree grammar is said to be in* NF2 *if no production rule uses the union operator (denoted by "+") in its content model.* □

Example 4. The regular tree grammar G_1 is not in NF2. The content model $(Author1^*, Pub, (Library + Museum))$ uses the union operator and occurs in a production rule. □

Any given regular tree grammar can be converted to a regular tree grammar in NF2. The conversion algorithm for this is more involved than for NF1, and is given in Table 4. The algorithm uses a function migrateUnion(RE). migrateUnion(RE) is a recursive function which takes as input any regular expression RE, and returns an equivalent regular expression RE' of the form $(RE_1 + RE_2 + \ldots + RE_n)$, where no RE_i, $1 \leq i \leq n$ contains the union operator.

Example 5. Converting G_1 to NF2, we obtain the regular tree grammar G_5 given in Table 5. □

Mapping between XML and other models has become very important in recent years, with several applications exporting their data to XML, and several applications storing the XML data they obtain using other data models. NF2 provides the basis for mapping union types provided by XML schemas to types in the target model. It is useful when the target model does not support union types, such as the relational model. An overview of how this conversion works is as follows: the type definitions as provided by NF2 are mapped into type definitions in the target language. For example, G_1 that uses union types is converted using NF2 to G_5 which does not have union types. In G_5, there are three production rules with $Book$ on the left-hand side. In the target language, there is a new type defined for each of these production rules, and thus there will be three types defined corresponding to $Book$, say $Book_1, Book_2, Book_3$. The language maintains additional book-keeping, this book-keeping assists in the mapping of operations from the original type definitions to the new ones. For example, a

Convert a regular tree grammar $G = (N, T, S, P)$ to NF2.

1. If there does not exist any rules in G of the form $R : X \rightarrow a\ RE$, where RE has the union operator, return G. Else go to Step 2.
2. Let migrateUnion$(RE) = (RE_1 + RE_2 \ldots RE_n)$.
3. Replace R with $\{X \rightarrow a\ RE_1, X \rightarrow a\ RE_2, \ldots, X \rightarrow a\ RE_n\}$, and go to Step 1.

migrateUnion : $RE \Longrightarrow RE'$, where $RE' = (RE_1 + RE_2 + \ldots + RE_n)$, and no RE_i, $1 \leq i \leq n$ contains the "+" operator

1. If RE does not contain "+", return RE.
2. If $RE = (r)^*$:
   Let migrateUnion$(r) = (r_1 + r_2 + \ldots + r_n)$.
   return $(r_1^*, r_2^*, \ldots, r_n^*)^*$.
3. If $RE = (r_1 + r_2)$:
   Let migrateUnion$(r_1) = (a_1 + a_2 + \ldots + a_m)$.
   Let migrateUnion$(r_2) = (b_1 + b_2 + \ldots + b_n)$.
   return $(a_1 + a_2 + \ldots + a_m + b_1 + b_2 + \ldots + b_n)$.
4. If $RE = (r_1, r_2)$:
   Let migrateUnion$(r_1) = (a_1 + a_2 + \ldots + a_m)$.
   Let migrateUnion$(r_2) = (b_1 + b_2 + \ldots + b_n)$.
   return $((a_1, b_1) + (a_1, b_2) + \ldots + (a_1, b_n) +$
   $(a_2, b_1) + (a_2, b_2) + \ldots + (a_2, b_n) + \ldots$
   $(a_m, b_1) + (a_m, b_2) + \ldots + (a_m, b_n))$.

Table 4: RTG to NF2 algorithm.

query such as $Book/Pub$ on the original XML schema will be mapped to, say, $(Book_1/Pub \cup Book_2/Pub \cup Book_3/Pub)$ in the new schema.

5 Mapping an XML Schema to Relational Schema

In this section, we describe our algorithm to map a given XML schema to a relational schema. Before we proceed, let us define *XSchema*, a language independent formalism to specify XML schemas. *XSchema* is based on regular tree grammar theory that we studied in the previous sections, and borrows from our definitions in [12, 13]. To define *XSchema*, we first assume the existence of a set $\widehat{E}$ of element names, a set $\widehat{A}$ of attribute names and a set $\widehat{\tau}$ of atomic data types defined in [1] (e.g., ID, IDREF, IDREFS, string, integer, date, etc). When needed, an attribute name $a \in \widehat{A}$ or an element name $e \in \widehat{E}$ is qualified by the element names using the *path expression* notation $e_1.e_2 \cdots e_n.a$, or $e_1.e_2 \cdots e_n.e$ where $e_i \in \widehat{E}, 1 \leq i \leq n$). *XSchema* extends regular tree grammars with the specification of data types, attribute definitions, primary key constraints, and inclusion dependency constraints. Further attributes of types IDREF and IDREFS identify the target types referred to by the values.

	N	=	$\{Book, Author1, Author2, Pub,$ $Library, Museum, Son, Daughter\}$
	T	=	$\{book, author, publisher,$ $library, museum, son, daughter\}$
	S	=	$\{Book\}$
P :	$Book$	$\rightarrow$	$book(Author1^*, Pub, Library)$
	$Book$	$\rightarrow$	$book(Author1^*, Pub, Museum)$
	$Book$	$\rightarrow$	$book(Author2^*, Pub, Library)$
	$Author1$	$\rightarrow$	$author(Son^*)$
	$Author2$	$\rightarrow$	$author(Daughter^*)$
	Pub	$\rightarrow$	$publisher(\epsilon)$
	$Library$	$\rightarrow$	$library(\epsilon)$
	$Museum$	$\rightarrow$	$museum(\epsilon)$
	Son	$\rightarrow$	$son(\epsilon)$
	$Daughter$	$\rightarrow$	$daughter(\epsilon)$

Table 5: G_5 = NF2 representation for G_1

.

Definition 4 (*XSchema*) *An XSchema is denoted by 6-tuple $\mathbb{X} = (E, A, M, P, r, \Sigma)$, where:*

- *E is a finite set of element names in $\widehat{E}$,*
- *A is a function from an element name $e \in E$ to a set of attribute names $a \in \widehat{A}$,*
- *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: $\alpha ::= \epsilon \mid \tau \mid \alpha + \alpha \mid \alpha, \alpha \mid \alpha^* \mid \alpha^? \mid \alpha^+$, where ϵ denotes the empty element, $\tau \in \widehat{\tau}$, "+" for the union, "," for the concatenation, "α^*" for the Kleene star, $\alpha^?$ for $(\alpha + \epsilon)$ and α^+ for (α, α^*),*
- *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 (τ, n, d, f), where $\tau \in \widehat{\tau}$, 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. Further more, if τ is* IDREF *or* IDREFS*, then τ also specifies the target type or types that the attribute value should refer to using the symbol "$\rightsquigarrow$",*
- *$r \subseteq E$ is a finite set of root elements,*
- *Σ is a finite set of integrity constraints for XML model. The integrity constraints we consider are primary key constraints and inclusion dependencies.* □

Example 6. We shall use for this section the following *XSchema* and XML document. This schema represents a conference, and is slightly modified from the one in [10]. It serves as a good example for explaining the various steps in mapping an XML schema to a relational schema. The *XSchema* is given by $\mathbb{X}_6 = (E, A, M, P, r, \Sigma)$, where

$$
\begin{aligned}
E &= \{conf, title, date, editor, \\
&\quad paper, contact, author, person, \\
&\quad name, email, phone, cite\} \\
A(conf) &= \{id\} \\
M(conf) &= (title, date, editor^?, paper^*) \\
P(conf.id) &= (\mathsf{ID}, \neg?, \epsilon, \epsilon) \\
M(title) &= (string) \\
A(date) &= \{year, mon, day\} \\
M(date) &= \epsilon \\
A(editor) &= \{eids\} \\
M(editor) &= (person^*) \\
P(eids) &= (\mathsf{IDREFS} \rightsquigarrow (person^*), ?, \epsilon, \epsilon) \\
A(paper) &= \{id\} \\
M(paper) &= (title, contact^?, author, cite^?) \\
P(paper.id) &= (\mathsf{ID}, \neg?, \epsilon, \epsilon) \\
A(contact) &= \{aid\} \\
M(contact) &= \epsilon \\
P(aid) &= (\mathsf{IDREF} \rightsquigarrow person, \neg?, \epsilon, \epsilon) \\
M(author) &= (person^*) \\
A(person) &= \{id\} \\
M(person) &= (name, (email + phone)^?) \\
P(person.id) &= (\mathsf{ID}, \neg?, \epsilon, \epsilon) \\
A(name) &= \{fn, ln\} \\
M(name) &= \epsilon \\
P(fn) &= (string, ?, \epsilon, \epsilon) \\
P(ln) &= (string, \neg?, \epsilon, \epsilon) \\
M(email) &= (string) \\
M(phone) &= (string) \\
A(cite) &= \{id, format\} \\
M(cite) &= (paper^*) \\
P(cite.id) &= (\mathsf{ID}, \neg?, \epsilon, \epsilon) \\
P(format) &= (string, ?, (ACM|IEEE), \epsilon) \\
r &= \{conf, paper\}
\end{aligned}
$$

$$
\Sigma = \{title, date.year \overset{key}{\rightarrow} conf, title \overset{key}{\rightarrow} paper, name.ln \overset{key}{\rightarrow} person\}
$$

□

An XML document conforming to the above schema is:

```
<conf id="er05">
  <title>Int'l Conf on Conceptual Modeling</title>
  <date>
    <year>2005</year> <mon>Nov</mon> <day>25</day>
  </date>
  <editor eids="sheth bossy">
    <person id="klavans">
      <name fn="Judith" ln="Klavans"/>
      <email>klavans@cs.columbia.edu</email>
    </person>
  </editor>
  <paper id="p1">
    <title>Indexing Model for Structured ...</title>
    <contact aid="dao"/>
    <author>
      <person id="dao">
        <name fn="Tuong" ln="Dao"/>
      </person>
    </author>
  </paper>
  <paper id="p2">
    <title>Logical Information ...</title>
    <contact aid="shah"/>
    <author>
      <person id="shah">
        <name fn="Kshitij" ln="Shah"/>
      </person>
      <person id="sheth">
        <name fn="Amit" ln="Sheth"/>
        <email>amit@cs.uga.edu</email>
      </person>
    </author>
    <cite id="c100" format="ACM">
      <paper id="p3">
        <title Making Sense of Scientific ...</title>
        <author>
          <person id="bossy">
            <name fn="Marcia" ln="Bossy"</name>
            <phone>391.4337</phone>
          </person>
        </author>
      </paper>
    </cite>
  </paper>
</conf>
<paper id="p7">
  <title>Constraints Preserving ...</title>
  <contact aid="lee"/>
  <author>
    <person id="lee">
      <name fn="Dongwon" ln="Lee"/>
      <email>dongwon@cs.ucla.edu</email>
    </person>
  </author>
  <cite id="c200" format="IEEE"/>
</paper>
```

There are several steps in mapping an XML schema to a relational schema: (a) schema simplification, where we obtain from a given *XSchema*, a simpler *XSchema*, which will not have the constraints that cannot be captured in the relational model (b) inlining of elements and attributes, where we use simple "heuristics" to determine what are the attributes of a relation (c) mapping collection types, collection types can be represented in *XSchema* using * or $^+$. For example, in $M(A) = (\ldots, B^*, \ldots)$, A defines collection type of B. Collection types are considered as 1:many relationships and are mapped as such, (d) mapping IDREF and IDREFS attributes, IDREFS attributes are treated similar to child elements, (e) capturing the order specified

in the XML model, and (f) enforcing constraints such as key constraints and inclusion dependencies.

Without loss of generality, we will assume for the rest of the section that the given *XSchema* is in NF2. We shall represent the element type definitions in the *XSchema* in NF2 as $(r_1 + r_2 + \ldots + r_n)$, rather than writing them out as multiple element type definitions. (This is equivalent to performing NF2, and then performing NF1 on the resulting schema.)

Example 7. When we represent $\mathbb{X}_6$ in NF2, the element type definitions change as: $M(conf) =$
$((title, data, paper^*) + (title, data, editor, paper^*))$
$M(paper) = ((title, author) + (title, contact, author) +$
$(title, author, cite) + (title, contact, author, cite))$
$M(person) = ((name) + (name, email) + (name, phone))$
□

5.1 Schema Simplification

As mentioned before, the relational model cannot capture all the constraints specified in the *XSchema*. Our schema simplification step is based on the following principle: *For a parent-child relationship or* IDREFS *attribute, we capture only the cardinality of the child with respect to the parent as can be expressed using the two-tuple* $[minOccur, maxOccur]$. *We do not capture any other constraint that may be specified in the XSchema.*

For example, consider the element type definition $M(A) = (D, (B, C, B)^*)$. We do not capture: (a) the order constraint that a C must be followed and preceded by a B, (b) that the number of Bs must be twice the number of Cs, or (c) that the number of Bs should be even. We simplify the above content model as $M'(A) = (D, B^*, C^*)$. The simplification rules are applied to the target types identified by IDREFS attributes also.

Consider an element e, whose element type definition, after NF2 is given by $M(e) = (r_1 + r_2 + \ldots + r_n)$. We apply the schema simplification rules given below to every sub-expression r_i. Remember that NF2 ensures that r_i will contain only "," and "*" operators. We express the occurrence constraints using the notation $[minOccur, maxOccur]$ for convenience. This representation is equivalent to the occurrence constraints specifiable using regular expressions.

Reg Exp	Simplified Reg Exp
$(r_1, r_2)[m, M]$	$(r_1[m, M], r_2[m, M])$
$(r_1[m_1, M_1])[m_2, M_2]$	$r_1[m_1 * m_2, M_1 * M_2]$
$(r_1[m_1, M_1], ..., r_1[m_2, M_2])$	$r_1[m_1 + m_2, M_1 + M_2]$

Table 6: Schema Simplification Rules

The *XSchema* after NF2 for $\mathbb{X}_6$ is already simplified, and the simplification rules described above do not change the schema.

5.2 Inlining

Inlining is used to generate more "meaningful" and efficient relational schemas. In inlining, we consider attributes of descendants of an element as attributes in the relation corresponding to that element. For example, consider the element *conf* which has child *date* which in turn has attributes *year*, *month*, and *day* in $\mathbb{X}_6$. Now we can inline the attributes of *date* to *conf* and obtain the relation $conf(year, month, date)$.

Inlining for an element e is done recursively using the function inline $(currEl, currSet, attSet)$ described in Table 7. inline returns a set of relations that should be generated for an input element $currEl$. The function also takes as input $currSet$ which denotes the current set of relations we have, and $attSet$ which is used to maintain the list of attributes of e that should be present in every relation generated for e.

To inline the element e, we call inline, where the initialization is: $currEl = e$, $currSet = \phi$, $attSet = \phi$.

inline : $currEl, currSet, attSet \Longrightarrow ResultSet$

1. Assign the set of attributes in $A(currEl)$ except IDREF and IDREFS attributes to $attSet$. Let the element type definition of $currEl$ be given by $M(currEl) = (r_1 + r_2 + \ldots + r_k)$. Set $ResultSet = \phi$.
2. For each r_i, we do the following.
   - 2.1. Set $currSet = attSet$.
   - 2.2. Let the elements which occur in r_i with occurrence constraint $[1, 1]$ after simplification be $\{e_1, e_2, \ldots, e_n\}$. For each e_i, do the following.
     - 2.2.1. If $M(e_i) \in \widehat{\tau}$, then $currSet = currSet \times e_i$.
     - 2.2.2. Else $currSet = currSet \times$ inline (e_i, ϕ, ϕ)
   - 2.3. If $currSet = \phi$, $currSet = currEl$.
   - 2.4. $ResultSet = ResultSet \cup currSet$.
3. return $ResultSet$.

Table 7: Inline Function

A *ResultSet* is returned by inline, and we define a relation corresponding to each term in the *ResultSet*. Note that when a subexpression r_i in $M(currEl)$ yields the empty set, we add the element name $(currEl)$ as a placeholder. This ensures that inlining results in two relations for the element *conf* in $\mathbb{X}_6$ as shown below.

Example 8. Suppose we perform inlining on *conf*, *paper*, and *person*, we obtain the following relation definitions. Note that inline for $M(editor)$ and $M(contact)$ actually produce the empty set, however inline will return *editor* or *contact* as placeholders.

conf: $conf1(id, title, year, mon, day)$,
$conf2(id, title, year, mon, day, editor)$.
paper: $paper1(id, title, author)$,
$paper2(id, title, contact, author)$,

$paper3(id, title, author, cite.id, format)$,
$paper4(id, title, contact, author, cite.id, format)$

person: $person1(id, fn, ln)$,
$person2(id, fn, ln, email)$,
$person3(id, fn, ln, phone)$ □

It is important to note that inlining could result in a huge number of relations. For instance, consider inlining an element a which has three choices, that is $M(a) = (a_1 + a_2 + a_3)$. Let each of these choices in turn have three more choices and so on, that is, $M(a_1) = (a_{11} + a_{12} + a_{13})$. Let the height of this be n. The number of relations for this element will be 3^n. For example, if the height is 2, then we get nine relations. This could result in "inefficient" relational models. In a later subsection, we will study different cases when we place restrictions, so that we do not create so many relations. Also, our inlining is used for non-recursive elements. Recursive elements are handled in a later subsection.

5.3 Mapping Collection Types

Relational model cannot specify collection types. An element type definition such as $M(book) = (author^*)$ is captured in the relational model by defining separate relations for *book* and *author*, and a foreign key attribute for *author* referencing *book*. We use the same technique, however with some modifications. The modifications become necessary because we can have two different type definitions which specify collection type of the same sub-element. For example, consider $M(book) = (author^*)$, and $M(article) = (author^*)$. We will create two relations for *author*, one with a foreign key referencing *book* and another with a foreign key referencing *article*. We may even have an element type definition as $M(book) = (author^* + (author^*, publisher))$. When we do inlining for *book*, we end up with two relations, say $book1(book)$ and $book2(publisher)$. In this case again we create two relations for *author*, one referencing $book1$ and the other referencing $book2$. The algorithm for the conversion of collection types is as follows.

Let an element A have multiple occurrences in element type definitions, $m_1, m_2, \ldots, m_n$, where for each m_i, A has multiple occurrences in sub-expressions for which the tables created are $m_{i1}, m_{i2}, \ldots, m_{in}$. Now a separate relation is created for A for each of these tables in each rule. Further, the table created for A corresponding to the table m_{ij} will define a foreign key referencing m_{ij}.

Example 9. In $\mathbb{X}_6$, we have $paper^*$ in the element type definition for *conf* and *cite*. *paper* may occur as the root element also. Therefore, we have the following table definitions for *paper*: $paper1$, $paper2$, $paper3$ and $paper4$ represent papers that occur at the root of the document. In addition, we define papers that occur as children of conferences by defining all possible combinations of $\{paper1, paper2, paper3, paper4\}$ with $\{conf1, conf2\}$. For example, three of the tables that are defined are:

$paper1conf1(id, title, author, conf.title, year)$,
$paper1conf2(id, title, author, conf.title, year)$,
$paper2conf1(id, title, contact, author, conf.title, year)$

Here, *paper* and *cite* form a recursive relationship and this is handled in a different subsection. Just like for *paper* and *conf*, we define *person* which can occur as children of *editor* or author. □

5.4 Handling IDREF and IDREFS attributes

Let us first consider IDREF attributes. An IDREF attribute specification will be of the form $B \rightarrow (@a :: \mathsf{IDREF} \rightsquigarrow (E_1 + E_2 + \ldots E_n))$, where $B \in E$, and E_i's either belong to E, or can be ϵ. Let the tables defined for any E_i be $m_{i1}, m_{i2}, \ldots, m_{in_i}$. Let the set of tables defined for B be $b_1, b_2, \ldots, b_n$. Our mapping replaces the set of tables for B with the set which is the cross product of $\{b_1, b_2, \ldots, b_n\}$ with the set of m_{ij}'s. Also for each of the resulting tables, we will have a foreign key referencing the ID attribute of m_{ij}.

Example 10. In $\mathbb{X}_6$, we have an IDREF attribute defined for *contact*, which refers to *person*. The set of tables defined for *contact* are $\{paper2, paper4\}$. Also we have three tables defined for *person* as $\{person1, person2, person3\}$. The result of our mapping is replacing $paper2$ and $paper4$ with the following six tables.

$paper2person1(id, title, person1, author)$
$paper2person2(id, title, person2, author)$
$paper2person3(id, title, person3, author)$
$paper4person1(id, title, person1, author, cite.id, cite.format)$
$paper4person2(id, title, person2, author, cite.id, cite.format)$
$paper4person3(id, title, person3, author, cite.id, cite.format)$

Here $person1$ column in each of the above tables is a foreign key referencing the *id* attribute of relation $person1$, $person2$ column references the *id* attribute of relation $person2$ and so on. □

IDREFS attributes are handled using the techniques for handling IDREF attributes and collection types, and techniques such as schema simplification and inlining. An IDREFS attribute specification is of the form $B \rightarrow (@a :: \mathsf{IDREFS} \rightsquigarrow RE)$, where $B \in E$, and RE is a regular expression over E. After schema simplification, let RE be given by $r = (r_1 + r_2 + \ldots + r_n)$. Now we perform a modified inlining, which we call IDREFSinline and is given in Table 8. IDREFSinline is

a non-recursive function, and we call it as IDREFSinline (r).

IDREFSinline : $RE \Longrightarrow ResultSet$

1. Let $RE = (r_1 + r_2 + \ldots + r_n)$. For each r_i in $\{r_1, r_2, \ldots, r_n\}$, do the following.
   - 1.1. Let the elements which occur in r_i with occurrence constraint $[1,1]$ (after simplification) be $S = \{e_1, e_2, \ldots, e_n\}$.
   - 1.2. If $S = \phi$, $ResultSet = ResultSet \cup currEl$.
   - 1.3. Else $ResultSet = ResultSet \cup (e_1, e_2, \ldots, e_n)$.
2. return $ResultSet$.

Table 8: Inline Function for IDREFS attributes

Let the tables for B currently defined be $\{b_1, b_2, \ldots, b_n\}$. After we do IDREFSinline on an attribute of B, the set of tables for B change as follows. Suppose $ResultSet$ returns $\{a_1, a_2, \ldots, a_n\}$. Let a_i be $(e_{i1}, e_{i2}, \ldots, e_{in_i})$. Let the set of tables for e_{ij} be denoted by the set T_{ij}. Now in $ResultSet$, we replace a_i with the set $T_{i1} \times T_{i2} \times \ldots \times T_{in_i}$. Now we replace the set of tables for B with the cross product of the sets $\{b_1, b_2, \ldots, b_n\}$ and $ResultSet$. Also in each of these tables, we define every column from $ResultSet$, say t_{ijk} as a foreign key referencing the ID attribute of the table t_{ijk}.

Now, consider an element say E that occurs as a collection type in r, we define a new set of tables, one for every possible combination of the tables of B and the tables of E. Each table has a set of columns representing the table in B, say b_i, which will be a foreign key referencing the primary key of b_i, and a column representing the table of E, say e_i, which will be a foreign key referencing the ID attribute of e_i.

Example 11. In $\mathbb{X}_6$, we have an IDREFS attribute defined for *editor* as $@eids :: \mathsf{IDREFS} \rightsquigarrow person^*$. The IDREFS attribute has a collection type, so we define a set of new tables, one for every combination of $\{conf2\}$ and $\{person1, person2, person3\}$ as follows.

$eidsconf2person1(title, year, person1id)$
$eidsconf2person2(title, year, person2id)$
$eidsconf2person3(title, year, person3id)$ □

5.5 Handling Recursion

Recursion can occur in *XSchema* in two ways: through a cycle of elements with optional "?" occurrence, or through a cycle of elements with "*" occurrence. For example, $A \rightarrow (@a, A^?)$ forms a recursion of the first kind. We will handle this type of recursion using inlining. However simple inlining would result in infinitely many number of relations - we will create separate relations for recursion of depth 0, 1, 2, and so on. The relations will be $A1(@a)$, $A2(@a, @a)$, $A3(@a, @a, @a)$ and so on. Therefore, instead, we will use foreign keys - we will create relations of the form $A(@a, ARef)$, where $ARef$ refers to the A element that occurs as child of this element. For example consider the XML document fragment

```
<A a='a1'>
   <A a='a2'/>
</A>
```

When we translate this, we get the tuple, $(a1, a2ref)$ where $a2ref$ refers to $a2$. However, simply creating foreign keys also results in infinitely many number of relations. We will create separate relations for the A element which has no more children as $A1(@a)$, the A element that will have $A1$ as a child, say $A2(@a, A1Ref)$, the A element that has $A2$ as a child, say $A3(@a, A2Ref)$, and so on. The above happens because we try to create many relations trying to enforce that we will not have null values in the relations. The solution is to create only one relation and allow null values for the columns. For example, the above relation will be translated to $A(@a, ARef)$ where $ARef$ is a foreign key referencing the relation A, and it can have null values.

The second kind of recursion is handled in a similar manner to how we handle collection types. For example, consider $A \rightarrow (@a, A^*)$. Our simple translation of collection types will again produce infinitely many relations – relations for As that do not have A as parents $A1(@a)$, As that have $A1$ as parent $A2(@a, A1Ref)$, As that have $A2$ as parent $A3(@a, A2Ref)$ and so on. This problem is solved again by enforcing one relation for A, as $A(@a, ARef)$, where $ARef$ is a foreign key referring to A and it can be null.

The general technique for handling recursion is similar to the one mentioned in [15]. For every strongly connected component, at least one of the elements must be defined as a separate relation. We enforce that such an element be mapped to exactly one relation. This is illustrated in Example 12. Further more, in a strongly connected component, if there exists an element which can be children of more than one element in the strongly connected component, then we define a separate relation for that element. This is necessary as we will see in Example 13.

Example 12. There is a cycle in $\mathbb{X}_6$ – formed by *paper*, and *cite*. This refers to the papers that are cited in a given paper. A separate relation is to be created for *paper*, which was already done. We enforce that there be only one relation corresponding to *paper*. Now we have to append all the attributes in the different tables corresponding to *paper* into one table. Doing this, we get one *paper* table as

$paper(id, title, person1, person2, person3, cite.id,$
$format, conf1.title, conf1.year, conf2.title,$
$conf2.year, @paperRef).$

Here $person1$, $person2$, $person3$ are nullable foreign keys, representing the contact author for a *paper*,

cite.id, $format$ are nullable and represent the cite information, and @$paperRef$ is a nullable foreign key that refers to a paper that cited this paper. □

Example 13. Consider a simple 4 element schema, defined as $A \rightarrow (@a, B)$, $B \rightarrow (@b, A?, C?)$, $C \rightarrow (@c, D)$, $D \rightarrow (@d, A?, C?)$. This forms one strongly connected component with two nodes A and C having in-degree greater than one. Consider a sample document represented for convenience as

$a1 \rightarrow b1 \rightarrow a2 \rightarrow b2 \rightarrow a3 \rightarrow b3 \rightarrow c1 \rightarrow d1 \rightarrow c2 \rightarrow d2 \rightarrow c3 \rightarrow d3 \rightarrow a4 \rightarrow b4 \rightarrow c4 \rightarrow d4$

Here $x \rightarrow y$ means that y is a child of x. Also ai means an element A, with the value of the attribute @a as ai.

Suppose we define only one relation, as there is only one strongly connected component. Let us try to define the relation for A. We will see that we can have multiple C, D elements for one A, and this cannot be captured in the relation for A. This happens because there is a C, D loop, rather C has two parents, B and D. In this case, we define another relation for C. The two relations are given below.

A

@a	@b	aR	cR
$a1$	$b1$	$a2$	*null*
$a2$	$b2$	$a3$	*null*
$a3$	$b3$	*null*	$c1$
$a4$	$b4$	*null*	$c4$

C

@c	@d	aR	cR
$c1$	$d1$	*null*	$c2$
$c2$	$d2$	*null*	$c3$
$c3$	$d3$	$a4$	*null*
$c4$	$d4$	*null*	*null*

Table 9: Relations from translation of Example 13. Here aR and cR are foreign keys referencing A and C respectively.

5.6 Capturing Order Specified in the XML model

It is necessary to maintain the order in which the elements occur in the document, so that we can reconstruct the document. Order can be captured in the XML model by keeping an order attribute corresponding to each element. The attribute which we maintain will give the position of the node in the whole document. However, capturing order is not the focus of this paper. So we will ignore it for the rest of this paper.

5.7 Capturing Semantic Constraints

XML schemas specify relationships using parent-child relationships, ID-IDREF/(S) attributes, and using inclusion dependencies. We have studied the mapping of all these features except inclusion dependencies. Inclusion dependencies are mapped as follows: Suppose the *XSchema* defines an inclusion dependency as: $A(X) \subseteq B(Y)$. Let the set of tables defined for B be $\{b_1, b_2, \ldots, b_n\}$, and the set of tables for A be $\{a_1, a_2, \ldots, a_n\}$. We replace the set of tables for A with the set formed by the cross product of $\{a_1, a_2, \ldots, a_n\}$ and $\{b_1, b_2, \ldots, b_n\}$. For any resulting table, say $a_i.b_j$, the set of columns is the same as the set of columns in a_i. Further, we define the inclusion dependency $a_i.b_j(X) \subseteq b_j(Y)$.

Key constraints are translated as follows. There are four kinds of tables that can be generated during our translation.

- *Table created corresponding to an element, the key for the element does not depend on any attribute of any of its ancestors*:
  Let the key for the element be denoted by K. K may consist of attributes, elements that are not collection types, elements that are collection types, IDREF attributes, and IDREFS attributes. First, we remove the elements that are collection types and the collection type elements in IDREFS attributes. Then, we replace every element with its *key*. Let the resulting key be K'. Corresponding to every attribute in K', there will be a column defined for the table. The key for the table is defined as this set of columns corresponding to K'.

- *Table created corresponding to an element, the key for the element depends on at least one attribute of its ancestors.*
  This occurs for "relative keys" [3]. Relative keys can be explained with this example, consider a library schema with rules as $library \rightarrow (book^*)$, $book \rightarrow (@title, author^*)$, $author \rightarrow (@name)$. Let the key for *book* be (@$title$). We could define the key for *author* as - for every *book*, the key for *author* is (@$name$). Here, the key for *author* can be considered as $(@name, book)$. Now translation of the keys is just like in the previous step. Therefore in the relational schema, we get the key for *book* as (@$title$), and the key for *author* as $(@name, bookRef)$.

- *Table created corresponding to collection type in* IDREFS *attributes*
  Consider $A \rightarrow (@a, @bRefs :: IDREFS \rightsquigarrow (B^*))$, $B \rightarrow (@b :: \mathsf{ID})$. For a collection type of an IDREFS attribute, a new table is created. Therefore we create a new table $bRefsAB(ARef, BRef)$, here $ARef$ is a foreign key referencing the primary key of A, and $BRef$ is a foreign key referencing the ID attribute of B. The key for this table will be the set of all columns of this table.

- *Table created corresponding to an element for which there is no key defined.*
  In this case, a system-generated identifier is introduced as the key for the table.

Example 14. In $\mathbb{X}_6$, for every table corresponding to $conf$, the key is $(title, year)$, for every table corresponding to $paper$, the key is $(title)$, and for every table corresponding to $person$, the key is (ln). Furthermore a set of tables are created corresponding to the IDREFS attribute $eids$, and the key for this is $(title, year, person)$. □

5.8 Decreasing the Number of Relations Generated

We have so far been generating multiple tables for an element. Also to handle recursion, only one table is created for an element in the recursion. Creating more tables decreases the number of null values in the resulting relation. However, often times the number of relations generated could be too many. So it is useful to restrict the number of relations produced for an element, though this will increase the number of null values. One extreme is enforcing that we create at most one table corresponding to an element. Or we can say that for every element that has a foreign key defined on it, we create only one table. We can also ask for user input, or use data and query statistics and iteratively improve the schema to give the "best" performance. Now, we will convert the example *XSchema*, $\mathbb{X}_6$ and the document to the relational model. We will assume that for an element that has a foreign key defined on it, we will create at most one table.

5.9 Example

When we convert $\mathbb{X}_6$ and the document using our mapping algorithm, we obtain the following set of relations. We do not capture the order among elements in the document. Also for an element that has a foreign key defined on it, we will create at most one table. Therefore we will have only one table for $conf$, $paper$ and $person$, as shown in Table 10. The constraints are:

$title, year \overset{key}{\rightarrow} conf$; $title \overset{key}{\rightarrow} paper$;
$ln \overset{key}{\rightarrow} person$; $title, year, ln \overset{key}{\rightarrow} eids$;
$eids.ln \subseteq person.ln$;
$eids.title, eids.year \subseteq conf.title, conf.year$;
$paper.aid \subseteq person.id$;
$paper.conf.title, paper.year \subseteq conf.title, conf.year$;
$paper.paperRef \subseteq paper.title$;
$person.conf.title, person.year \subseteq conf.title, conf.year$;
$person.paper.title \subseteq paper.title$

5.10 Analysis of our algorithm

Our first set of experiments were to determine the "goodness" of the relational schema we obtain. Our example illustrated that our conversion algorithm produces good relational schemas. We then took the TPC-H data and converted that to XML using the CoT algorithm [12]. Now we ran our XML to relational conversion on this XML data. We obtained the original relational data that we started off with.

We then analyzed the properties of our algorithm closely, and proved that when we convert the XML data resulting from CoT to relations using the techniques described in this paper, we will obtain the original relational schema that we started with. These results are quite promising for a common application scenario - person A wants to ship his data in a relational database to person B. A converts his data to XML and then ships this XML data. B receives this XML data, and converts it back to relations and stores it in his relational database. Now if A uses CoT for his relational to XML conversion, and if B uses the steps described in this paper for XML to relational conversion, then it is guaranteed that B will end up with the same relations as A started off with, no additional "handshake" between A and B is required.

6 Conclusion

In this paper, we presented a theoretical basis for different XML applications using regular tree grammars. We defined the NF1 representation, which is useful in document validation as well as schema validation. Our NF2 representation forms the basis for mapping type definitions in XML schemas to a language that does not provide union types, such as SQL. Further, we described the steps in mapping an XML schema to a relational schema, while maintaining semantic constraints. Prelimiary studies indicate that our algorithm generates "good" relational schema, and further it complements our relational to XML conversion algorithm, CoT.

There are still several issues to be studied with respect to data modeling using XML schemas. One interesting question is what restrictions can be imposed on the XML schemas for data modeling. For example, ability to specify recursive types is one of the advantages of the XML data model, but CoT generates XML schemas that have no recursion. Also there are several important questions regarding operations for the XML model. What is a set of "good" operations for the XML model? How do we map operations on the XML model to the relational model? Answering these questions will clarify the data modeling aspects of XML schemas.

References

[1] P. V. Biron and A. Malhotra (Eds). "XML Schema Part 2: Datatypes". W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-2/.

[2] P. Bohannon, J. Freire, P. Roy, and J. Simeon. "From XML Schema to Relations: A Cost-Based Approach to XML Storage". In *IEEE ICDE*, San Jose, CA, Feb. 2002.

eids

title	year	ln
Int'l Conf...	2005	sheth
Int'l Conf...	2005	bossy

conf

id	title	year	mon	day
er05	Int'l Conf...	2005	Nov	25

paper

id	title	aid	cite.id	format	conf.title	year	paperRef
p1	Indexing Model...	dao	null	null	Int'l Conf...	2005	null
p2	Logical Information...	shah	c100	ACM	Int'l Conf...	2005	null
p3	Making Sense...	null	null	null	null	null	Logical Information...
p7	Constraints Preserving...	lee	c200	IEEE	null	null	null

person

id	fn	ln	email	phone	conf.title	year	paper.title
klavans	judith	klavans	klavans@cs.columbia.edu	null	Int'l Conf...	2005	null
dao	Tuong	Dao	null	null	null	null	Indexing Model...
shah	Kshitij	Shah	null	null	null	null	Logical Information...
sheth	Amit	Sheth	amit@cs.uga.edu	null	null	null	Logical Information...
bossy	Marcia	Bossy	null	391.4337	null	null	Making Sense...
lee	Dongwon	Lee	dongwon@cs.ucla.edu	null	null	null	Constraints Preserving...

Table 10: Relations from translation of $\mathbb{X}_6$ and the example document.

[3] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. "Keys for XML". In *Int'l World Wide Web Conf. (WWW)*, Hong Kong, May 2001.

[4] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu (Eds). "XQuery 1.0: An XML Query Language". W3C Working Draft, Jun. 2001. http://www.w3.org/TR/2001/WD-xquery-20010607/.

[5] J. Clark and M. Murata (Eds). "RELAX NG Tutorial". OASIS Working Draft, Jun. 2001. http://www.oasis-open.org/committees/relax-ng/tutorial.html.

[6] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. "Tree Automata Techniques and Applications", 1997. http://www.grappa.univ-lille3.fr/tata.

[7] A. Deutsch, M. F. Fernandez, and D. Suciu. "Storing Semistructured Data with STORED". In *ACM SIGMOD*, Philadephia, PA, Jun. 1998.

[8] D. Florescu and D. Kossmann. "Storing and Querying XML Data Using an RDBMS". *IEEE Data Eng. Bulletin*, 22(3):27–34, Sep. 1999.

[9] H. Hosoya and B. C. Pierce. "XDuce: A Typed XML Processing Language". In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, May 2000.

[10] D. Lee and W. W. Chu. "CPI: Constraints-Preserving Inlining Algorithm for Mapping XML DTD to Relational Schema". *J. Data & Knowledge Engineering (DKE)*, 39(1):3–25, Oct. 2001.

[11] D. Lee, M. Mani, F. Chiu, and W. W. Chu. "Nesting-based Relational-to-XML Schema Translation". In *Int'l Workshop on the Web and Databases (WebDB)*, Santa Barbara, CA, May 2001.

[12] D. Lee, M. Mani, F. Chiu, and W. W. Chu. "NeT & CoT: Translating Relational Schemas to XML Schemas using Semantic Constraints". Technical report, UCLA Computer Science Dept., Feb. 2002.

[13] M. Mani, D. Lee, and R. D. Muntz. "Semantic Data Modeling using XML Schemas". In *Int'l Conf. on Conceptual Modeling (ER)*, Yokohama, Japan, Nov. 2001.

[14] M. Murata, D. Lee, and M. Mani. "Taxonomy of XML Schema Languages using Formal Language Theory". In *Extreme Markup Languages*, Montreal, Canada, Aug. 2001. http://www.cs.ucla.edu/∼dongwon/paper/.

[15] 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.

[16] M. Takahashi. "Generalizations of Regular Sets and Their Applicatin to a Study of Context-Free Languages". *Information and Control*, 27(1):1–36, Jan. 1975.

[17] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn (Eds). "XML Schema Part 1: Structures". W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-1/.