Constraints-preserving Transformation from XML Document Type Definition to Relational Schema*

Dongwon Lee and Wesley W. Chu

Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
{dongwon,wwc}@cs.ucla.edu

Abstract. As Extensible Markup Language (XML) [5] is emerging as *the* data format of the internet era, there are increasing needs to efficiently store and query XML data. One way towards this goal is using relational database by transforming XML data into relational format. In this paper, we argue that existing transformation algorithms are not complete in the sense that they focus only on *structural* aspects and ignoring *semantic* aspects. We present the semantic knowledge that needs to be captured during the transformation to ensure a correct relational schema. Further, we show a simple algorithm that can 1) derive such semantic knowledge from the given XML Document Type Definition (DTD) and 2) preserve the knowledge by representing them in terms of *semantic constraints* in relational database terms. By combining the existing transformation algorithms and our *constraints-preserving* algorithm, one can transform XML DTD to relational schema where correct semantics and behaviors are guaranteed by the preserved constraints. Experimental results are also presented.

1 Introduction

As the World-Wide Web becomes a major means of disseminating and sharing information, Extensible Markup Language (XML) [5] is emerging as a possible candidate data format because it is simpler than SGML and more powerful than HTML. To query XML data, one way is to reuse the established relational database techniques by converting and storing XML data in relational storage. Since the hierarchical XML and the flat relational data models are not fully compliant, the transformation is not a straightforward task.

To this end, several XML-to-relational transformation algorithms have been studied [8, 9, 16]. Although they work well for the given applications, to a greater or lesser extent, they miss one important point. That is, the transformation algorithms only capture the *structure* of the Document Type Definition (DTD) and

* This research is supported in part by DARPA contract No. N66001-97-C-8601.

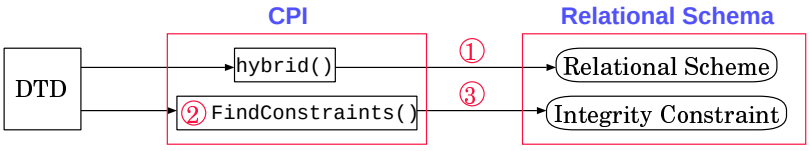

Fig. 1. Overview of our approach. Numbers 1) to 3) specify: 1) transforming schema, 2) discovering constraints via `FindConstraints()`, and 3) preserving constraints via `RewriteConstraints()`.

ignore the *semantic* constraints hidden in it. In this paper, via our *constraints-preserving inlining (CPI)* algorithm, we show the kinds of semantic constraints that can be derived from DTD during transformation, and how to preserve them by re-writing them in resulting schema notation. Since our algorithm to capture and preserve semantic constraints from DTD is orthogonal to transformation algorithms, ours can be applied to various transformation algorithms in [8, 9, 16] with little change. Figure 1 presents an overview of our approach. First, given a DTD, we transform it to a corresponding relational scheme using an existing algorithm. Second, during the transformation, we discover various semantic constraints in XML notation. Third, we re-write the discovered constraints to conform to relational notation.

This paper is organized as follows. Section 2 gives background information and related work. In Section 3, one transformation algorithm is discussed in detail. Section 4 presents various semantic constraints that are hidden in DTD. Section 5 proposes our algorithm to preserve such constraints during transformation. Section 6 reports some experimental results that we have conducted and Section 7 summarizes with concluding remarks.

2 Background and Related Work

Relational Schema: We define a relational schema $\mathcal{R}$ to be composed of a *relational scheme* ($\mathcal{S}$) and *semantic constraints* (Δ). That is, $\mathcal{R} = (\mathcal{S}, \Delta)$. In turn, the relational scheme $\mathcal{S}$ is a collection of table schemes such as $r(a_1, ..., a_k)$, where a_i is the i-th attribute in the table r and the semantic constraints Δ is a collection of various semantic knowledge such as domain constraints, inclusion dependency, equality-generating dependency, tuple-generating dependency, etc.

XML and DTD: XML is a textual representation of the hierarchical data that is being defined by the World-Wide Web Consortium [5]. The meaningful piece of the XML document is bounded by matching starting and ending *tags* such as `<name>` and `</name>`. In XML, tags are defined by users while in HTML, permitted tags are pre-defined. Thus, XML is a meta-language that can be used for defining other customized languages. Using DTD, users can define the structure of the XML document of particular interest. A DTD in XML is very similar to a schema in a relational database. The main building blocks of DTD are *elements*

Table 1. A DTD for `Conference`.

```
<!DOCTYPE Conference [
  <!ELEMENT conf     (title,date,editor?,paper*)>
  <!ATTLIST conf     id      ID          #REQUIRED>
  <!ELEMENT title    (#PCDATA)>
  <!ELEMENT date     EMPTY>
  <!ATTLIST date     year    CDATA       #REQUIRED
                     mon     CDATA       #REQUIRED
                     day     CDATA       #IMPLIED>
  <!ELEMENT editor   (person*)>
  <!ATTLIST editor   eids    IDREFS      #IMPLIED>
  <!ELEMENT paper    (title,contact?,author,cite?)>
  <!ATTLIST paper    id      ID          #REQUIRED>
  <!ELEMENT contact  EMPTY>
  <!ATTLIST contact  aid     IDREF       #REQUIRED>
  <!ELEMENT author   (person+)>
  <!ATTLIST author   id      ID          #REQUIRED>
  <!ELEMENT person   (name,(email|phone)?)>
  <!ATTLIST person   id      ID          #REQUIRED>
  <!ELEMENT name     EMPTY>
  <!ATTLIST name     fn      CDATA       #IMPLIED
                     ln      CDATA       #REQUIRED>
  <!ELEMENT email    (#PCDATA)>
  <!ELEMENT phone    (#PCDATA)>
  <!ELEMENT cite     (paper*)>
  <!ATTLIST cite     id      ID          #REQUIRED
                     format (ACM|IEEE)   #IMPLIED>
]>
```

and *attributes*, which are defined by the keywords `<!ELEMENT>` and `<!ATTLIST>`, respectively. In general, components in DTD are specified by the following BNF syntax:

```
<!ELEMENT> <element-name> <element-type>
<!ATTLIST> <attr-name> <attr-type> <attr-option>
```

For a detailed description of DTD model, refer to [12]. Table 1 shows a DTD for `Conference` which states that a `conf` element can have four sub-elements: `title`, `date`, `editor` and `paper` in that order. As common in regular expression, 0 or 1 occurrence (i.e., *optional*) is represented by the symbol "`?`", 0 or more occurrences is represented by the symbol "`*`", and 1 or more occurrences is represented by the symbol "`+`". A sub-element without any such symbols (e.g., `title`) represents a *mandatory* one.

Keywords `#PCDATA` and `CDATA` are used as *string* types for elements and attributes, respectively. For instance, the type of `title` element is defined as `#PCDATA` so that `title` element can be arbitrary character data. `<attr-option>` can be `#REQUIRED` or `#IMPLIED` among others. An attribute with a `#REQUIRED` option is a *mandatory* one while an attribute with a `#IMPLIED` option is an *optional* one. `<attr-type>` keywords `ID` and `IDREF` are used for the pointed and pointing attributes, respectively. `IDREFS` is a plural form of `IDREF`. For instance, the `author` element must have a mandatory `id` attribute and this attribute is used when other attributes point to this attribute. On the other hand, the `contact` element has a mandatory `aid` attribute that must point to the `id` attribute of the contacting `author` of the current paper. One interesting definition in Table 1 is the `cite` element; it can have zero or more `paper` elements as sub-elements, thus

Table 2. A valid XML document conforming to the DTD for `Conference` in Table 1.

```
<conf id="er99">
  <title>Int'l Conference on Conceptual Modeling (ER)</title>
  <date> <year>1999</year> <mon>May</mon> <day>20</day> </date>
  <editor eids="sheth bossy">
    <person id="klavans">
      <name fn="Judith" ln="Klavans"/><email>klavans@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 Modeling of Heterogeneous...</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 Information...</title>
        <author>
          <person id="bossy">
            <name fn="Marcia" ln="Bossy"/><phone>391.4337</phone>
          </person>
        </author> </paper> </cite> </paper>
</conf>
<paper id="p7">
  <title>Constraints-preserving Transformation from XML...</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>
```

creating a cyclic definition. Table 2 shows a valid XML document conforming to the DTD for `Conference`. The document represents a portion of the fictional ER conference held in 1999. The first two `paper` elements are described with `id="p1"` and `id="p2"`, respectively. The `paper` element with `id="p2"` further has a `cite` element that describes the references in the paper. The `paper` element with `id="p7"` shows an example of the valid XML document that is *not* rooted at `conf` element. Note that a valid XML document can be rooted at any level of the DTD hierarchy as long as their sub-elements and attributes follow the DTD syntax.

Assumptions: Without loss of generality, to simplify our presentation, we assume that: 1) the input DTD has been already simplified using a technique in [16], 2) the input XML documents are all *valid*, and 3) the XML features such as *entities* or *notations* are not covered.

Related Work: [16] presents three transformation algorithms that focus on the table level of the schema while [9] studies different performance issues among eight algorithms that focus on the attribute and value level of the schema. Since our CPI algorithm provides a systematic way of finding and preserving con-

straints from a DTD, ours is an improvement to the existing transformation algorithms. Work done in STORED [8] deals with *non-valid* XML documents. When input XML documents do not conform to the given DTD, STORED uses a data mining technique to find a representative DTD whose support exceeds the pre-defined threshold. Since our algorithm to find and preserve constraints is not directly tied to a single transformation algorithm, ours can be applied to this algorithm as well. [13] also presents a DTD inference algorithm when it is not known. [4] discusses template language-based transformation from XML DTD to relational schema which requires human experts to write an XML-based transformation rule.

Some work has been done in [17] dealing with the transformation from relational tables to XML documents. There has been some transformation work in the OODB area as well [6]. Since OODB is a richer environment than RDB, their work is not readily applicable to our application. The logical database design methods and their associated transformation techniques to other data models have been extensively studied in ER research. For instance, [3] presents an overview of such techniques. However, due to the differences between ER and XML models, those transformation techniques need to be modified substantially.

3 Transforming DTD to Relational Schema

Transforming a hierarchical XML model to a flat relational model is not a trivial task. There are several difficulties including non 1-to-1 mapping, set values, recursion, and fragmentation issues [16]. For a better presentation, we chose one particular transformation algorithm, called the *hybrid inlining algorithm* [16] among many algorithms [4, 8, 9, 16]. It is chosen since it exhibits the pros of the other two competing algorithms in [16] without severe side effects and it is a more generic algorithm than those in [4, 8]. Since issues of discovering and preserving semantic constraints in this paper is orthogonal to that of transformation algorithms, our technique can be applied to other transformation algorithms easily.

3.1 Hybrid Inlining Algorithm

The *hybrid* algorithm [16] essentially does the following[1]:

1. Create a *DTD graph* that represents the structure of a given DTD. A DTD graph can be constructed when parsing the given DTD. Its nodes are elements, attributes, or operators in DTD. Each element appears exactly once in the graph, while attributes and operators appear as many times as they appear in the DTD.

[1] We have made a few changes to the hybrid algorithm for a better presentation (e.g., renaming, supporting "|" operator), but the crux of the algorithm remains intact.

2. Sub-elements in the choice model using the operator "|" are treated as if they are in the ordered sequence model with the following changes: 1) "+" operator is converted to "*" operator, 2) sub-elements without any occurrence operators are appended by "?" operator. For instance, `<!ELEMENT A ((a|b)+|c)>` is converted to `<!ELEMENT A (a*,b*,c?)>`. Further, an attribute with `#IMPLIED` or `IDREFS` type is converted to an operator node "?" or "+" in a DTD graph.
3. Identify *top nodes* in a DTD graph. A top node satisfies any of the following conditions: 1) not reachable from any nodes (e.g., source node), 2) direct child of "*" or "+" operator node, 3) recursive node with indegree > 1, or 4) one node between two mutually recursive nodes with indegree $= 1$. Then, starting from a top node T, *inline* all the elements and attributes at *leaf nodes* reachable from T unless they are other top nodes.
4. Attribute names are composed by the concatenated path from the top node to the leaf node using "_" as a delimiter. Use an attribute with `ID` type as a key if provided. Otherwise, add a system-generated integer key[2].
5. If a table corresponds to the shared element with indegree > 1 in DTD, then add a field `parent_elm` to denote the parent element to which the current tuple belongs. Further, for each shared element, a new field `fk_$X$` is added as a *foreign key* to record the key values of parent element X. If X is inlined into another element Y, then record the Y's key value in the `fk_$Y$` field.
6. Inlining an element Y into a table r corresponding to another element X (i.e., top node) creates a problem when an XML document is rooted at the element Y. To facilitate queries on such elements, a new field `root_elm` is added to a table r.
7. If an *ordered* DTD model is used, a field `ordinal` is added to record position information of sub-elements in the element. (For simplification, the `ordinal` field is not shown in this paper.)

Table 3 shows the output of the transformation by the hybrid algorithm.

Among eleven elements in the DTD in Table 1, four elements – `conf`, `paper`, `person`, and `eids` – are top nodes and thus chosen to be mapped to the different tables. For the top node `conf`, the elements `date`, `title`, and `editor` are reachable and thus inlined. Then, the `id` attribute is used as a key and the `root_elm` field is added. For the top node `paper`, the elements `title`, `contact_aid`, `author`, `cite_format` and `cite_id` are reachable and inlined. Since the `paper` element is shared by the `conf` and `cite` elements (two incoming edges in a DTD graph), new fields `parent_elm`, `fk_conf` and `fk_cite` are added to record who and where the parent node was. Note that in the `paper` table (Table 3), a tuple with `id="p7"` has the value `"paper"` for the `root_elm` field. This is because the element `<paper id="p7">` is rooted in the DTD (Table 2) without being embedded in other elements. Consequently, its `parent_elm`, `fk_conf` and `fk_cite` fields are null. For the top node `person`, the elements `name_fn`, `name_ln` and `email` are reachable

[2] In practice, even if there is an attribute with `ID` type, one may decide to have a system-generated key for better performance.

Table 3. A relational scheme ($\mathcal{S}$) along with the associated data that are converted from the DTD in Table 1 and XML document in Table 2 by the hybrid algorithm. Note that the hybrid algorithm does not generate *semantic constraints (Δ)*.

conf

id	root_elm	title	date_year	date_mon	date_day
er99	conf	ER	1999	May	20

conf_editor_eids

id	root_elm	fk_conf	eids
100001	conf	er99	sheth
100002	conf	er99	bossy

paper

id	root_elm	parent_elm	fk_conf	fk_cite	title	contact_aid	cite_id	cite_format
p1	conf	conf	er99	–	Indexing ...	dao	–	–
p2	conf	conf	er99	–	Logical ...	shah	c100	ACM
p3	conf	cite	–	c100	Making ...	–	–	–
p7	paper	–	–	–	Constraints ...	lee	c200	IEEE

person

id	root_elm	parent_elm	fk_conf	fk_paper	name_fn	name_ln	email	phone
klavans	conf	editor	er99	–	Judith	Klavans	klavans@cs...	–
dao	conf	paper	–	p1	Tuong	Dao	–	–
shah	conf	paper	–	p2	Kshitij	Shah	–	–
sheth	conf	paper	–	p2	Amit	Sheth	amit@cs...	–
bossy	conf	paper	–	p3	Marcia	Bossy	–	391.4337
lee	paper	paper	–	p7	Dongwon	Lee	dongwon@cs...	–

and inlined. Since the `person` is shared by the `author` and `editor` elements, again, the `parent_elm` is added. Note that in the `person` table (Table 3), a tuple with `id="klavans"` has the value `"editor"`, not `"paper"`, for the `parent_elm` field. This implies that "klavans" is in fact an `editor`, not an `author` of the paper.

4 Semantic Constraints in DTD

Domain Constraints When the domain of the attributes is restricted to a certain specified set of values, it is called *Domain Constraints*. For instance, in the following DTD, the domain of the attributes `gender` and `married` are restricted.

```
<!ATTLIST author gender  (male|female) #REQUIRED
                 married (yes|no)      #IMPLIED>
```

In transforming such DTD into relational schema, we can enforce the domain constraints using SQL `CHECK` clause as follows:

```
CREATE DOMAIN gender VARCHAR(10) CHECK (VALUE IN ("male", "female"))
CREATE DOMAIN married VARCHAR(10) CHECK (VALUE IN ("yes", "no"))
```

When the mandatory attribute is defined by the `#REQUIRED` keyword in DTD, it needs to be forced in the transformed relational schema as well. That is, the attribute `ln` cannot be omitted below.

```
<!ELEMENT person  EMPTY>
<!ATTLIST person  fn CDATA  #IMPLIED   ln CDATA  #REQUIRED>
```

We use the notation "$X \not\rightarrow \emptyset$" to denote that an attribute X cannot be null. This kind of domain constraint can be best expressed by using the `NOT NULL` clause in SQL as follows:

```
CREATE TABLE person (fn VARCHAR(20), ln VARCHAR(20) NOT NULL)
```

Cardinality Constraints In DTD declaration, there are only 4 possible cardinality relationships between an element and its sub-elements as illustrated below:

```
<!ELEMENT article (title, author+, reference*, price?)>
```

A. 1-to-{0,1} mapping ("at most" semantics): An element can have either zero or one sub-element. (e.g., sub-element `price`)
B. 1-to-{1} mapping ("only" semantics): An element must have one and only one sub-element. (e.g., sub-element `title`)
C. 1-to-{0, ...} mapping ("any" semantics): An element can have zero or more sub-elements. (e.g., sub-element `reference`)
D. 1-to-{1, ...} mapping ("at least" semantics): An element can have one or more sub-elements. (e.g., sub-element `author`)

For convenience, let us call each cardinality relationship as type A, B, C, and D, respectively. From these cardinality relationships, mainly three constraints can be inferred. First, whether or not the sub-element can be null. Similar to the attribute case, we use the notation "$X \not\rightarrow \emptyset$" to denote that an element X cannot be null. This constraint is easily enforced by the `NULL` or `NOT NULL` clause. Second, whether or not more than one sub-elements can occur. This is also known as *singleton constraint* in [18] and is one kind of equality-generating dependencies. Third, given an element, whether or not its sub-element should occur. This is one kind of tuple-generating dependencies. The second and third types will be further discussed below.

Inclusion Dependencies (IDs) An *Inclusion Dependency* assures that values in the columns of one fragment must also appear as values in the columns of other fragments and is a generalization of the notion of *referential integrity*.

Trivial form of IDs found in DTD is that "given an element X and its sub-element Y, Y must be included in X (i.e., $Y \subseteq X$)". For instance, from the `conf` element and its four sub-elements in DTD, the following IDs can be found as long as `conf` is not null: {`conf.title` $\subseteq$ `conf`, `conf.date` $\subseteq$ `conf`, `conf.editor`

$\subseteq$ `conf, conf.paper` $\subseteq$ `conf`}. Another form of IDs can be found in the attribute definition part of DTD with the use of the `IDREF(S)` keyword. For instance, consider the `contact` and `editor` elements in the DTD in Table 1 shown below:

```
<!ELEMENT person  (name,(email|phone)?>
<!ATTLIST person  id   ID     #REQUIRED>
<!ELEMENT contact EMPTY>
<!ATTLIST contact aid  IDREF  #REQUIRED>
<!ELEMENT editor  (person*)>
<!ATTLIST editor  eids IDREFS #IMPLIED>
```

The DTD restricts the `aid` attribute of the `contact` element such that it can only point to the `id` attribute of the `person` element[3]. Further, the `eids` attribute can only point to multiple `id` attributes of the `person` element. As a result, the following IDs can be derived: {`editor.eids` $\subseteq$ `person.id, contact.aid` $\subseteq$ `person.id` }. IDs can be best enforced by the "foreign key" concept if the attribute being referenced is a primary key. Otherwise, it needs to use the `CHECK`, `ASSERTION`, or `TRIGGERS` facility in SQL.

Equality-Generating Dependencies (EGDs) The *Singleton Constraint* [18] restricts an element to have "at most" one sub-element. When an element type X satisfies the singleton constraint towards its sub-element type Y, if an element instance x of type X has *two* sub-elements instances y_1 and y_2 of type Y, then y_1 and y_2 must be the same. This property is known as *Equality-Generating Dependencies (EGDs)* and denoted by "$X \rightarrow Y$" in database theory. For instance, two EGDs: {`conf` $\rightarrow$ `conf.title, conf` $\rightarrow$ `conf.date`} can be derived from the `conf` element in Table 1. This kind of EGDs can be enforced by SQL `UNIQUE` construct. In general, EGDs occur in the case of the 1-to-{0,1} and 1-to-{1} mappings in the cardinality constraints.

Tuple-Generating Dependencies (TGDs) *Tuple-Generating Dependencies (TGDs)* in relational model require that some tuples of a certain form be present in the table and use the "$\twoheadrightarrow$" symbol. Two useful forms of TGDs from DTD are the *child* and *parent constraints* [18].

1. **Child constraint:** `"Parent` $\twoheadrightarrow$ `Child"` states that every element of type *Parent* must have at least one child element of type *Child*. This is the case of the 1-to-{1} and 1-to-{1,...} mappings in the cardinality constraints. For instance, from the DTD in Table 1, since the `conf` element must contain the `title` and `date` sub-elements, the child constraint `conf` $\twoheadrightarrow$ {`title, date`} holds.

[3] Precisely, an attribute with `IDREF` type does not specify which element it should point to. This information is available only by human experts. However, new XML schema languages such as XML Schama and DSD can express where the reference actually points to [12].

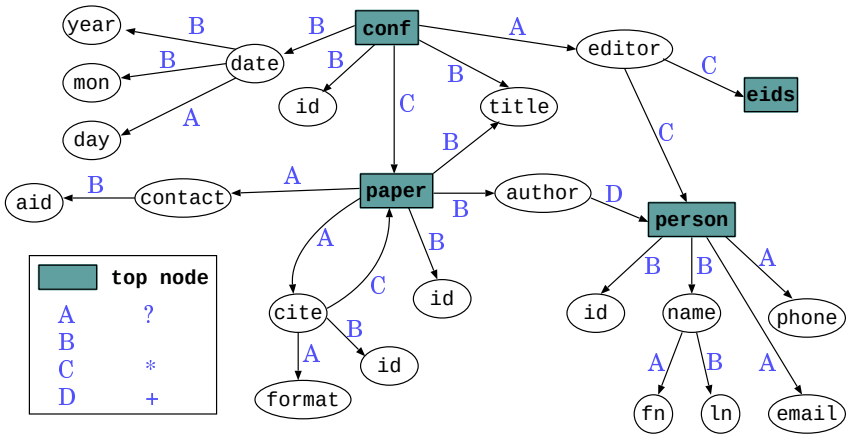

Fig. 2. An *Annotated DTD graph* for the `Conference` DTD in Table 1. The associated values of the nodes (i.e., indegree, type, tag, and status) are not shown.

2. **Parent constraint:** "`Child` $\twoheadrightarrow$ `Parent`" states that every element of type *Child* must have a parent element of type *Parent*. According to XML specification, there is no notion of *root* in DTD. That is, XML documents can start from any level of elements without necessarily specifying its parent element. Therefore, parent constraints cannot be assured simply by looking at the DTD structure. Rather, it requires some semantic knowledge. In the DTD in Table 1, for instance, the `editor` and `date` elements can have the `conf` element as their parent. Further, if we know that all XML documents were started at the `conf` element level rather than the `editor` or `date` level, then the parent constraint {`editor, date`} $\twoheadrightarrow$ `conf` holds. Note that the `title` $\twoheadrightarrow$ `conf` does not hold since the `title` element can be a sub-element of either the `conf` or `paper` element.

5 Discovering and Preserving Semantic Constraints

To help find semantic constraints, we use the following data structure:

Definition 1. *An* **annotated DTD graph (ADG)** $\mathcal{G}$ *is a pair* $(\mathcal{V}, \mathcal{E})$*, where* $\mathcal{V}$ *is a finite set and* $\mathcal{E}$ *is a binary relation on* $\mathcal{V}$*. The set* $\mathcal{V}$ *consists of element and attributes in a DTD. Each edge* $e \in \mathcal{E}$ *is labeled with the cardinality relationship types (A to D) as defined in Section 4. In addition, each vertex* $v \in \mathcal{V}$ *carries the following information:*

1. `indegree` *stores the number of incoming edges.*
2. `type` *contains the element type name in the content model of the DTD (e.g.,* `conf` *or* `paper`*).*
3. `tag` *stores a flag value whether the node is an element or attribute (if attribute, it contains the attribute keyword like* `ID` *or* `IDREF`*, etc.).*

Table 4. Cardinality relationships and their corresponding semantic constraints.

Relationship	Type	Symbol	Semantics	not null	EGDs	TGDs
1-to-{0,1}	A	?	at most	no	yes	no
1-to-{1}	B		only	yes	yes	yes
1-to-{0,...}	C	*	any	no	no	no
1-to-{1,...}	D	+	at least	yes	no	yes

4. `status` *contains "visited" flag if the node was visited in a depth-first search or "not-visited".*

Note that the cardinality relationship types in ADG considers not only element vs. sub-element relationships but also element vs. attribute relationships. For instance, from the DTD `<!ATTLIST X Y #IMPLIED Z #REQUIRED>`, two types of cardinality relationships (i.e., type A between element X and attribute Y, and type B between element X and attribute Z) can be derived. Figure 2 illustrates an example of ADG for the `Conference` DTD in Table 1. Then, the cardinality relationships can be used to find semantic constraints in a *systematic* fashion. Table 4 summarizes 3 main semantic constraints that can be derived from. The `FindConstraints()` algorithm can be immediately derived from the properties in Table 4. For detailed description, refer to [11].

Semantic constraints discovered by `FindConstraints()` have additional usage as we have shown in [11]. However, to enforce correct semantics in the newly generated relational schema, the semantic constraints in XML terms need to be re-written in relational terms. This is done by the algorithm `RewriteConstraints()`.

5.1 CPI: Constraints-preserving Inlining Algorithm

We shall now describe our complete DTD-to-relational schema transformation algorithm: *CPI (Constraints-preserving Inlining) algorithm* is a combination of the hybrid inlining, `FindConstraints()` and `RewriteConstraints()` algorithms. The CPI algorithm is illustrated in `CPI()` and `hybrid()`.

The algorithm first identifies all the top nodes from the ADG. This can be done using algorithms to find sinks or strongly-connected components in a graph [16]. Then, for each top node, the algorithm generates a corresponding table scheme using `hybrid()`. The associated constraints are found and re-written in relational terms using `FindConstraints()` and `RewriteConstraints()`, respectively. The `hybrid()` algorithm scans an ADG in a depth-first search manner while finding constraints and inlines a new field in the leaf node. The final output schema is the union of all the table schemes and semantic constraints.

Table 5 contains the semantic constraints that are re-written from XML terms to relational terms. As an example, the CPI algorithm will eventually spit out the following SQL `CREATE` statement for the `paper` table. Note that not only is the relational scheme provided, but the semantic constraints are also ensured by use of the `NOT NULL`, `KEY`, `UNIQUE` or `CHECK` constructs.

Algorithm 1: `RewriteConstraints`

Input : Constraints Δ' in XML notation
Output: Constraints Δ in relational notation

switch Δ' **do**
- **case** $X \not\rightarrow \emptyset$
  - If X is mapped to attribute X' in table scheme A, then $A[X']$ cannot be null. (i.e., "`CREATE TABLE` A (...X' `NOT NULL`...)") ;
- **case** $X \subseteq Y$
  - If X and Y are mapped to attributes X' and Y' in table scheme A and B, respectively, then re-write it as $A[X'] \subseteq B[Y']$. (i.e., If Y' is a primary key of B, then "`CREATE TABLE` A (...`FOREIGN KEY` (X') `REFERENCES` $B(Y')$...)". Else "`CREATE TABLE` A (...(X') `CHECK` (X' IN (SELECT Y' FROM B))...") ;
- **case** $X \rightarrow X.Y$
  - If element X and Y are mapped to the same table scheme A (i.e., since Y is not a top node, Y becomes an attribute of table A) and Z is the key attribute of A, then re-write it as $A[Z] \rightarrow A[Y]$. (i.e., "`CREATE TABLE` A (...`UNIQUE` (Y), `PRIMARY KEY` (Z)...)");
- **case** $X \twoheadrightarrow X.Y$
  - **if** (element X and Y are mapped to the same table) **then**
    - Let A be the table and Z be the key attribute of A. Then re-write it as $A[Z] \twoheadrightarrow A[Y]$. (i.e., "`CREATE TABLE` A (...Y `NOT NULL`, `PRIMARY KEY` (Z)...)") ;
  - **else**
    - Let the tables be A and B, respectively and Z be the key attribute of A. Then re-write it as $B[fk\_A] \subseteq A[Z]$. (i.e., "`CREATE TABLE` B (...`FOREIGN KEY` (fk_A) `REFERENCES` $A(Z)$...)")

return Δ;

```
CREATE TABLE paper (
  id          NUMBER      NOT NULL,
  title       VARCHAR(50) NOT NULL,
  contact_aid VARCHAR(20),
  cite_id     VARCHAR(20),
  cite_format VARCHAR(50) CHECK (VALUE IN ("ACM", "IEEE")),
  root_elm    VARCHAR(20) NOT NULL,
  parent_elm  VARCHAR(20),
  fk_cite     VARCHAR(20) CHECK (fk_cite IN (SELECT cite_id FROM paper)),
  fk_conf     VARCHAR(20),
  PRIMARY KEY (id),
  UNIQUE (cite_id),
  FOREIGN KEY (fk_conf)      REFERENCES conf(id),
  FOREIGN KEY (contact_aid)  REFERENCES person(id)
);
```

Algorithm 2: `CPI`

Input : Annotated DTD Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: Relational Schema $\mathcal{R}$

$\mathcal{V} \leftarrow$ `topnode`($\mathcal{G}$);
for *each* $v \in \mathcal{V}$ **do**
  $table\_def \leftarrow \{\}$;
  if $v.tag =$ *'element'* **then**
    `add`('root_elm', $table\_def$); `/* start where? */`
  if $v.indegree > 1$ **then**
    `add`('parent_elm', $table\_def$); `/* shared elements case */`
    `add`(concat('fk_', parent(v)), $table\_def$);
  $\mathcal{W} \leftarrow Adj[v]$; w $\in \mathcal{W}$;
  if *any* $w.tag =$ `'ID'` **then** `add`($w.type$, $table\_def$);
  else `add`('id', $table\_def$); `/* system-generated primary key */`
  $\mathcal{R} \leftarrow \mathcal{R} +$ `hybrid`(v, $table\_def$, $\emptyset$);
return $\mathcal{R}$;

Algorithm 3: `hybrid`

Input : Vertex v, TableDef $table\_def$, string $attr\_name$
Output: Relational Schema $\mathcal{R}$

$v.status \leftarrow$ 'visited';
for *each* $w \in Adj[v]$ **do**
  if $w.status =$ *'not-visited'* **then**
    $\Delta' \leftarrow$ `FindConstraints`(v, w);
    $\Delta \leftarrow$ `RewriteConstraints`(Δ');
    `hybrid`(w, $table\_def$, `concat`($attr\_name$, '_', $w.type$));
`add`($attr\_name$, $table\_def$); $\mathcal{R} \leftarrow table\_def + \Delta$;
return $\mathcal{R}$;

6 Experimental Results

We have implemented the CPI algorithm in Java using the IBM XML4J package. Table 6 shows a summary of our experimentation. We gathered test DTDs from "http://www.oasis-open.org/cover/xml.html" and [15]. Since some DTDs had syntactic errors caught by the XML4J, we had to modify them manually. Note that people seldom used the `ID` and `IDREF(S)` constructs in their DTDs except the `XMI` and `BSML` cases. The number of tables generated in the relational schema was usually smaller than that of elements/attributes in DTD due to the inlining effect. The only exception to this phenomenon was the `XMI` case, where extensive use of types C and D cardinality relationships resulted in many top nodes in the ADG.

The number of semantic constraints had a close relationship with the design of DTD hierarchy and the type of cardinality relationship used in the DTD. For instance, the `XMI` DTD had many type C cardinality relationships, which do

Table 5. The semantic constraints in relational notation for the `Conference` DTD in Table 1.

Type	Semantic constraints in relational notation
ID	conf_editor_eids[eids] $\subseteq$ person[id], paper[contact_aid] $\subseteq$ person[id]
EGD	conf[id] $\rightarrow$ conf[title,date_year,date_mon,date_day] paper[id] $\rightarrow$ conf[title,contact_aid,cite_id,cite_format] person[id] $\rightarrow$ conf[name_fn,name_ln,email]
TGD	conf[id] $\twoheadrightarrow$ conf[title,date_year,date_mon,date_day] paper[id] $\twoheadrightarrow$ conf[title,contact_aid,cite_id,cite_format] person[id] $\twoheadrightarrow$ conf[name_fn,name_ln,email], conf_editor_eids[fk_conf] $\subseteq$ conf[id] paper[fk_conf] $\subseteq$ conf[id], paper[fk_cite] $\subseteq$ paper[cite_id] person[fk_conf] $\subseteq$ conf[id], person[fk_paper] $\subseteq$ paper[id]
not null	conf[id,title,date_year,date_mon,root_elm] $\not\rightarrow \emptyset$ conf_editor_eids[id,root_elm] $\not\rightarrow \emptyset$ paper[id,title,root_elm] $\not\rightarrow \emptyset$, person[id,name_ln,root_elm] $\not\rightarrow \emptyset$

Table 6. Experimental results of the CPI algorithm.

DTD		DTD Schema				Relational Schema				
Name	Domain	Elm	Attr	ID	IDREF(S)	Table	Attr	$\rightarrow$	$\twoheadrightarrow$	$\not\rightarrow \emptyset$
`novel`	literature	10	1	1	0	5	13	6	9	9
`play`	Shakespeare	21	0	0	0	14	46	17	30	30
`tstmt`	religious text	28	0	0	0	17	52	17	22	22
`vCard`	business card	23	1	0	0	8	19	18	13	13
`ICE`	content syndication	47	157	0	0	27	283	43	60	60
`MusicML`	music description	12	17	0	0	8	34	9	12	12
`OSD`	s/w description	16	15	0	0	15	37	2	2	2
`PML`	web portal	46	293	0	0	41	355	29	36	36
`Xbel`	bookmark	9	13	3	1	9	36	9	1	1
`XMI`	metadata	94	633	31	102	129	3013	10	7	7
`BSML`	DNA sequencing	112	2495	84	97	104	2685	99	33	33

not contribute to the semantic constraints. As a result, the number of semantic constraints at the end was small compared to that of elements/attributes in DTD. This was also true for the `OSD` case. On the other hand, in the `ICE` case, since it used many type B cardinality relationships, it resulted in many semantic constraints. For detailed discussions on the experimentation and the implementation of the CPI algorithm, please refer to [10].

7 Conclusion

This paper presents a method to transform XML DTD to relatonal schema both in *structural* and *semantic* aspects. After discussing the semantic constraints hidden in DTD, two algorithms are presented for: 1) discovering the semantic

constraints using the hybrid inlining algorithm, and 2) re-writing the semantic constraints in relational notation. Our experimental results reveal that constraints can be systematically preserved during the conversion from XML to relational schema. Such constraints can also be used for semantic query optimization or semantic caching [11].

References

1. Abiteboul, S., Buneman, P., Suciu, D. "Data on the Web: From Relations to Semistructured Data and XML", *Morgan Kaufmann Publishers*, 2000.
2. Böhm, K., Aberer, K., Öszu, M. T., Gayer, K. "Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition", *Proc. IEEE Advances in Digital Libraries (ADL)*, Los Alamitos, California, April, 1998.
3. Batini, C., Ceri, S., Navathe, S. B. "Conceptual Database Design: An Entity-Relationship Approach", *The Benjamin/Cummings Pub. Inc.*, 1992.
4. Bourret, R. "XML and Databases", *Internet Document*, September, 1999. http://www.informatik.tu-darmstadt.de/DVS1/staff/bourret/xml/XMLAndDatabases.htm
5. Bray, T., Paoli, J., Sperberg-McQueen, C. M. (ed.), "Extensible Markup Language (XML) 1.0", *W3C Recommendation*, Feburary, 1998.
6. Christophides, V., Abiteboul, S., Cluet, S., Scholl, M. "From Structured Document to Novel Query Facilities", *Proc. ACM SIGMOD*, Minneapolis, Minnesota, 1994.
7. Deutsch, A., Fernandez, M. F., Florescu, D., Levy, A., Suciu, D. "XML-QL: A Query Language for XML", *Proc. The Query Language Workshop (QL)*, 1998. http://www.w3.org/TR/NOTE-xml-ql
8. Deutsch, A., Fernandez, M. F., Suciu, D. "Storing Semistructured Data with STORED", *Proc. ACM SIGMOD*, Philadephia, Pennsylvania, June, 1998.
9. Florescu, D., Kossmann, D. "Storing and Querying XML Data Using an RDBMS", *IEEE Data Engineering Bulletin*, 22(3), September, 1999.
10. "XPRESS Home Page", 2000. http://www.cobase.cs.ucla.edu/projects/xpress/
11. Lee, D., Chu, W. W. "Constraints-preserving Transformation from XML Document Type Definition to Relational Schema (Extended Version)", *UCLA-CS-TR 200001*, 2000. http://www.cs.ucla.edu/~dongwon/paper/
12. Lee, D., Chu, W. W. "Comparative Analysis of Six XML Schema Languages", *UCLA-CS-TR 200008*, 2000. http://www.cs.ucla.edu/~dongwon/paper/
13. Ludäescher, B., Papakonstantinou, Y., Velikhov, P., Vianu, V. "View Definition and DTD Inference for XML", *Proc. Post-ICDT Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, 1999.
14. Robie, J., Lapp, J., Schach, D. "XML Query Language (XQL)", *WWW The Query Language Workshop (QL)*, December, 1998.
15. Sahuguet, A. "Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask", *Proc. 3rd Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000.
16. Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., Naughton, J. "Relational Databases for Querying XML Documents: Limitations and Opportunities", *Proc. VLDB*, Edinburgh, Scotland, 1999.
17. Turau, V. "Making Legacy Data Accessible for XML Applications", *Internet Document*, 1999. http://www.informatik.fh-wiesbaden.de/~turau/veroeff.html
18. Wood, P. T. "Optimizing Web Queries Using Document Type Definitions", *Proc. 2nd Int'l Workshop on Web Information and Data Management (WIDM)*, 1999.