Evolution of Schema of XML-documents Stored in a Relational Database

Andrey Simanovsky

St Petersburg State University
Math&Mech, CS Dept.
asimanovsky@acm.org

Abstract. XML is today a standard for manipulating semistructured data. One of widely used industrial solutions, especially for systems with a fairly well defined data structure, is storing XML in a relational database, while XML queries are converted to SQL queries to the underlying relational database. A software product that produces XML "interface" to an underlying relational database commonly requires revision of XML and relational schemas with every new version of the product. Those schema and subsequent data transformations are selected and performed using *ad-hoc* algorithms. We propose a framework, namely formal evolution model, allowing semi-automatic schema transformation and data transformation for a new product version, thus discarding the necessity of *ad-hoc* algorithm design. The framework also allows a-priory estimating of query conversion performance.

Keywords. schema evolution

1 Introduction

The eXtensive Markup Language is today a de facto standard for manipulating semistructured data. This creates a set of data management challenges of storing and querying XML documents. One of widely engaged approaches is storing XML in a relational databases and providing query processor, which converts queries to XML documents to SQL queries to these underlying databases [6]. Even though industrial systems provide XML support and there are native XML databases, the "relational" approach dominates because it allows utilizing highly developed RDBMS technologies.

In many applications XML documents belonging to a particular domain comply with a particular schema designed for the domain. A number of algorithms of storing XML documents in relational databases, in particular those based on inlining [11], make use of the schema to provide better performance of the system.

Schemas used for particular domains tend to evolve in time, while many documents satisfying old schemas are already stored in relational database. E. g. it is a common practice for many software products to change both,

```
DTD99:
<!ELEMENT book (booktitle, author)>
<!ELEMENT article(title, author*, contactauthor)>
<!ATTLIST contactauthor authorID IDREF IMPLIED>
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor(monograph*)>
<!ATTLIST editor name CDATA #REQUIRED>
<!ELEMENT author (name, address)>
<!ELEMENT name(firstname?, lastname)>
DTD01:
<!ELEMENT book (booktitle, price?, author, authority*)>
<!ELEMENT authority (authname, country)>
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor(monograph+)>
<!ATTLIST editor name CDATA #REQUIRED>
<!ELEMENT author (name, address)>
<!ELEMENT name(firstname, lastname)>
```

Figure 1. Two DTDs for books taken from [9] (DTD99) and [11] (DTD01).

XML and relational schema, with every next version of the product, whereas the transformation algorithms for old versions of the database are developed *ad-hoc*. Such changes are often driven by new functionality requirements and, thereupon, regard performance as a minor question. Our example on Figure 1 models that case.

We argue that schema evolution is a framework that enables partial automation of the process of schema and data transformations and allows discarding of *ad-hoc* algorithms. We present a schema evolution model for a schema of XML documents that especially takes into account relational storage and introduces guidelines for the relational schema design as well.

There are two issues that a schema evolution problem consists of: the semantics of change, i. e., the changes of database structure, and change propagation, i. e., the transformation of existing data so that it complies with the new schema. The latter question was largely investigated in recent time ([10], [13], [15]) and has no specifics for XML documents stored in a relational database as compared to a general case. In this paper we concentrate on the former issue.

The model has many in common with traditional approach to evolution of object-oriented data. However, it is less restrictive than straight-forward application of techniques used for temporal object-oriented databases (i. e. those described in [1]).

1.1 Related work

Many research projects address storing XML in relational databases and providing efficient querying to the stored data. [5] and [8] provide an overview together with approximate performance estimates of main ideas underlying the majority of techniques of storing XML documents in a relational database for both situations: when the schema of the document is known and when it is not. [6], [9], [11] propose different algorithms based on shredding (storing) an XML document into a relational database with inlining method as well as algorithms of translation of queries over XML into relational queries. [12] discusses questions of storing order information for inlining methods and other algorithms of XML document shredding. However, these works do not address the issue of schema evolution concentrating rather on the performance questions. Our work utilizes the results obtained in these works considering them in the new aspect of schema evolution.

The schema evolution problem was largely investigated for temporal relational and object-oriented databases (e .g. in [1]). There were attempts to apply solutions for object-oriented databases to the evolution of schemas of XML documents. An evolution model and a classification of elementary schema transformations, with which it was suggested to describe general schema transformations, are proposed in [14]. However, presented evolution models often require multiple steps to acquire intuitively "elementary" schema changes. E. g. in our example swapping tags *contactauthor* and *authorid* may seem an atomic action while it would require three elementary operations to be performed on the schema if we employ general evolution model from [14].

1.2 Roadmap

In Section 2 we review the notion of a DTD-graph. In Section 3 we discuss DTD-graph factorizing. We use DTD-graph factorizing to build a lattice similar to the type lattice used when evolution of object-oriented databases is described. In Section 4 we present invariants for a factorized DTD-graph, and describe schema evolution in their terms. In Section 5 we describe elementary operations of the model and corresponding DTD transformations. In Section 6 our results are summarized and future research directions are outlined.

2 Building a DTD-graph

The process of building a DTD graph is explained in [11]. However, since its notion is slightly different in our work, and since it is essential for the understanding of the evolution model we present it in detail. A general DTD can be defined as follows:

Definition 2.1. *DTD definition.* Let C be a class of languages over an alphabet V. A DTD over V with respect to C is a mapping : $V \rightarrow C$.

Note that defining DTD as a mapping over a (necessarily) finite set allows describing DTD with a mapping table.

An example of C can be the class of regular languages, or a more sophisticated language, e. g. one used for DTDs defining XML document schemas. In the process of building a DTD-graph we consider a class of languages C_0 that can be defined with use of two constructs: a star ("*") and grouping ("(. . .)". We will call DTDs with respect to that class a simplified DTDs to distinguish them from DTDs used to describe real-life XML documents, which we will address to as regular DTDs.

The first stage of building a DTD-graph is obtaining a simplified DTD from a regular one. The process may be described as follows:

- The elements that are defined with <!ENTITY> tags are substituted with their definitions.

- The alphabet V is a union of tag and attribute names of the original DTD.
- Each <!ELEMENT> or <!ATTLIST> element of the DTD is considered as an entry in the mapping table that defines the simplified DTD.
- Words REQUIRED, IMPLIED etc are interpreted in terms of "*" and " " signs.
- Regular expression language over DTD tags used in ELEMENT tags is replaced with a language from the simplified DTDs language class using the rules presented in [11] and replacing all '|' with grouping.

DTD99:
<!ELEMENT book (booktitle, author)>
<!ELEMENT article(title, author*, contactauthor)>
<!ELEMENT contactauthor (authorID)>
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor(monograph*)>
<!ELEMENT author (name, address)>
<!ELEMENT name(firstname, lastname)>
DTD01:
<!ELEMENT book (booktitle, price, author, authority*)>
<!ELEMENT authority (authname, country)>
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor(monograph*)>
<!ELEMENT editor (name)>
<!ELEMENT author (name, address)>
<!ELEMENT name(firstname, lastname)>

Figure 2. Simplified DTDs for DTD99 and DTD01.

The simplified DTDs on Figure 2 correspond to the regular DTDs of the example described earlier.

Having a simplified DTD over an alphabet V with respect to a set of languages C_0 we define a DTD-graph as follows:

Definition 2.2. *DTD-graph.* DTD-graph is a graph (V, E) where V is the mentioned above set of attributes and $E \subseteq V \times V \times \{``*", ``\ "\}$

Vertices of the DTD-graph denote the tags and attributes of the original DTD. Edges of the DTD-graph denote the nesting relations between tags. The "*" mark appears on those edges that denote one-to-many nesting relationship. Figure 3 demonstrates a DTD-graph obtained for DTD99.

A number of algorithms (e. g., those in [12]) use similar notions of a DTD-graph extending it with additional information to provide a mapping of XML documents into relational database.

3 Introducing a lattice

At first we consider a factorization (normalization) of the DTD-graph. The following notions are used.

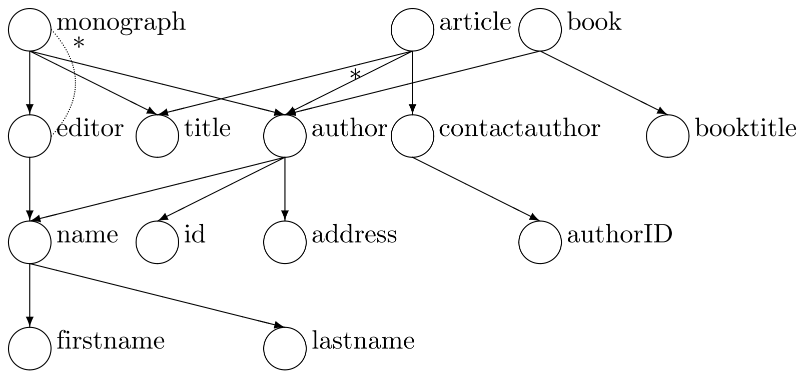

Figure 3. DTD-graph for DTD99.

Definition 3.1. *DTD-graph roots.* $Roots$ is an explicitly selected non-empty subset of DTD-graph ($Roots \subseteq V \ \wedge \ Roots \neq \emptyset$) vertices including all vertices without incoming edges: $\forall v \ In(v) = \emptyset \ \Rightarrow \ v \in Roots$

The roots of a DTD-graph are vertices corresponding to XML tags that can appear as outermost tags of an XML-document. We will assume that there is just one root where the existence of multiple roots is insignificant.

Definition 3.2. *Dominator.* A vertex of a DTD-graph is a dominator of a given vertex iff it appears on any path from any root to the given vertex. The domination relation is denoted as $a < b$ (denotes that a is a dominator of b).

Definition 3.3. *Immediate dominator.* Immediate dominator of the given vertex is a dominator of the given vertex, which is connected to the vertex with an edge without "*" mark on it.

Note that "*"-mark restriction makes our notion of immediate dominator more restrictive than the generally accepted term is.

Definition 3.4. *Attribute.* Attribute is a vertex without outgoing edges.

Definition 3.5. *Factorized DTD-graph.* Factorized DTD-graph G is a 4-ple (V', E', A, α) where $V' \subseteq V^*$ and the following is true:

- factorized vertex sets are disjoint : $V_1 \neq V_2 \in V' \ \Rightarrow \ V_1 \cap V_2 = \emptyset$;
- no "*"-marked edges are present "inside" a factorized vertex : $((v_1, v_2), sign) \in E \ \wedge \ \exists V_0 \mid v_1, v_2 \in V_0 \Rightarrow \ sign = \text{“ ''}$;
- sets are normalized with respect to the domination relation $<$: $v_1, v_2 \in V_0 \ \wedge \ v_1 < v_2 \ \Rightarrow \ \forall v \in In(v_1) \, v < v_2$;
- E' is a multi-set of pairs from $V' \times \{\text{“}*\text{''}, \text{“ ''}\}$;
- set of attributes A : $A = \{v \mid Out(v) = \emptyset\}$;
- vertex attributes α is a mapping : $V' \rightarrow A^*$ such that $\alpha(V_0) \subseteq V_0$.

Note that G can be created from a DTD-graph by merging vertices with their immediate dominators. G is obtained as soon as there are no more vertices to merge.

Many algorithms build relational schema on the basis of a DTD-graph. The factorization technique described largely follows the method used in [11] to construct relational schema. We do not assume that the created sets of attributes are the relations that are stored in a database, though we expect that the attributes of G correspond to attributes of the relational schema. I. e., results of the mapping α are views over the relational schema, and the stored procedures that calculate the views are stored in the database.

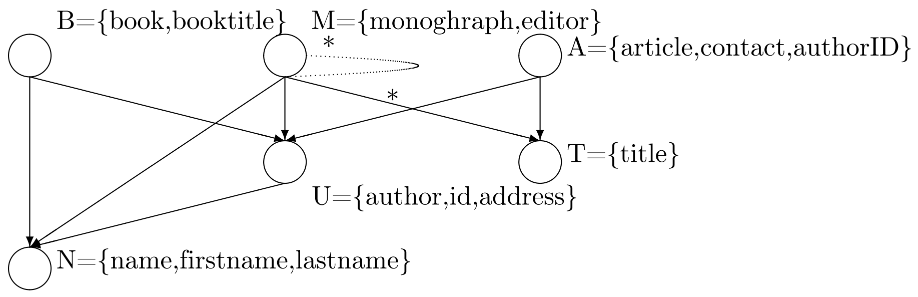

Figure 4. Factorized DTD-graph for DTD99.

A factorized DTD-graph for our working example is presented on Figure 4.

4 Factorized DTD-graph invariants

We suggest describing schema evolution model through a set of invariants applied to G. A natural way to formalize restrictions on the graph would be requiring that selected sets of vertices or attributes should (not) appear on each (or any, especially if an original DTD contained '|' rather than grouping) (acyclic) path from any root to the given vertex. These restrictions are invariants (axioms) that any accepted schema should comply with.

In the case of acyclic DTD-graph a possible set of invariants was presented for object-oriented databases in [1]. This set of axioms is presented on Figure 5. It can be applied with slight terminology changes. The following notions is used.

Definition 4.1. *Immediate predecessors.* The hierarchy on V' is defined by the sets of immediate predecessors $P(t)$ for each vertex t. Immediate predecessors are vertices that explicitly include vertex t and do not include t indirectly (through a transitive inclusion).

Definition 4.2. *Essential predecessors.* Essential predecessors $P_e(t)$ are explicitly specified sets of vertices. It is required that $P(t) \subset P_e(t)$.

Definition 4.3. *Vertex subhierarchy.* Vertex subhierarchy $PL(t)$ is a sub-graph of G, which vertex set consists of vertices, from which t is reachable.

1. Closure: $\forall t \in V'\ P_e(t) \subseteq V'$
2. Acyclicity: $\forall t \in V'\ t \notin \cup\alpha_x(PL(x), P(t))$
3. Rootedness: $\exists T \in V'\ \forall t \in V'\ T \in PL(t) \wedge P_e(T) = \emptyset$
4. Pointedness: $\exists \perp V'\ \forall t \in V'\ t \in PL(\perp)$
5. Immediate predecessors:
   $\forall t \in V'\ P(t) = P_e(t) - \cup\alpha_x(PL(x) \cap P_e(t) - \{x\}, P_e(t))$
6. Predecessors graph: $\forall t \in V'\ PL(t) = \cup\alpha_x(PL(x), P(t)) \cup \{t\}$
7. Interface: $\forall t \in V'\ I(t) = N(t) \cup H(t)$
8. Nativeness: $\forall t \in V'\ N(t) = N_e(t) - H(t)$
9. Inheritance: $\forall t \in V'\ H(t) = \cup\alpha_x(I(x), P(t))$

Figure 5. Axiom set for acyclic case.

Definition 4.4. *Native attributes.* Native attributes $N(t)$ are a set of attributes defined in vertex t.

Definition 4.5. *Inherited attributes.* Inherited attributes $H(t)$ are the union of attributes of all its predecessors.

Definition 4.6. *Essential attributes.* Essential attributes $N_e(t)$ are explicitly specified set of attributes. It is required that $N(t) \subset N_e(t)$.

Definition 4.7. *Interface.* Interface $I(t)$ is the union of its inherited and native attributes.

$N_e(t)$ denotes attributes (i. e. leaf tags or attributes of original DTD), which values are "essential" for working with a vertex of G, P_e denotes "essential" vertices (i. e. multiple occurences of a tag in original DTD).

Existence of root T and terminating vertex $\perp$ can be replaced with existence of final sets of vertices with the same properties. That would not significantly alter any results, so we retain the form of the corresponding axioms as they were stated in [1].

Vertices with single incoming edge are allowed in the evolution model. In that case we assume that G has a forward-edge ending in a vertex with a single incoming edge or the edge is marked with "*". If several possibilities for a given vertex exist we allow any — evolution model does not specify, which one to select.

To handle cyclic factorized DTD-graphs we introduce several additional notions.

Definition 4.8. *Immediate descendants.* The reverse hierarchy on V' is defined by the sets of immediate descendants $D(t)$ for each vertex t. Immediate descendants are vertices that explicitly are included into vertex t and are not included into t indirectly (through a transitive inclusion).

Definition 4.9. *Vertex reverse subhierarchy.* Vertex reverse subhierarchy $DL(t)$ is a sub-graph of G, which vertex set consists of vertices reachable from t.

Definition 4.10. *Essential descendants.* Essential descendants $D_e(t)$ are explicitly specified sets of vertices. It is required that $D(t) \subset D_e(t)$.

1. Closure: $\forall t \in V' \ P_e(t), D_e(t) \subseteq V'$
2. Rootedness: $\exists T \in V' \ \forall t \in V' \ T \in PL(t) \wedge P_e(T) = \emptyset$
3. Pointedness: $\exists \perp V' \ \forall t \in V' \ t \in PL(\perp)$
4. Immediate predecessors:
   $\forall t \in V' \ P(t) = P_e(t) - \cup\alpha_x(PL(x) \cap P_e(t) \cap (PL(t) - DL(t)) - \{x\}, P_e(t)) - \cup\alpha_x(PL(x) \cap P_e(t) \cap (PL(t) \cap DL(t)) - \{x\}, P_e(t)) - \cup\alpha_x((PL(x) - P(x)) \cap (PL(t) - DL(t)) \cap P_e(t), PL(x) \cap DL(x) - \{x\})$
5. Predecessors graph: $\forall t \in V' \ PL(t) = \cup\alpha_x(PL(x), P(t)) \cup \{t\}$
6. Immediate descendants:
   $\forall t \in V' \ D(t) = D_e(t) - \cup\alpha_x(DL(x) \cap D_e(t) \cap (DL(t) - PL(t)) - \{x\}, D_e(t)) - \cup\alpha_x(DL(x) \cap D_e(t) \cap (DL(t) \cap PL(t)) - \{x\}, D_e(t)) - \cup\alpha_x((DL(x) - D(x)) \cap (DL(t) - PL(t)) \cap D_e(t), DL(x) \cap PL(x) - \{x\})$
7. Descendants graph: $\forall t \in V' \ DL(t) = \cup\alpha_x(DL(x), D(t)) \cup \{t\}$
8. Interface: $\forall t \in V' \ I(t) = N(t) \cup H(t)$
9. Nativeness: $\forall t \in V' \ N(t) = N_e(t) - H(t)$
10. Inheritance: $\forall t \in V' \ H(t) = \cup\alpha_x(I(x), P(t))$

Figure 6. Axiom set for general case.

The axiom set for a general case is shown on Figure 6.

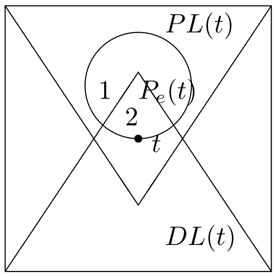

Figure 7. Understanding general case immediate predecessors axiom.

We introduce new axioms for controlling $D(t)$ and $DL(t)$ sets and modify axiom 4. The idea of modification is to avoid transitive inclusion to be "passed" through recursive vertices inclusion. The new axiom 4 states that $P(t)$ consists of those vertices from $P_e(t)$ that:

- do not lie in the area marked 2 (vertices of $P_e(t)$ that are both predecessors and descendants of t) on Figure 7 and transitively include t through a vertex from area 2,
- do not lie in the area marked 1 (vertices of $P_e(t)$ that are not descendants of t) on Figure 7 and transitively include t through a vertex from area 1,
- do not lie in area 1 and transitively include t through a vertex from area 2.

The calculation of $N(t)$, $I(t)$ and $H(t)$ sets requires calculation of $PL(t)$ set. Provided that initially all sets except essential ones are empty $PL(t)$ can be calculated by iterative application of the axioms. The following claim states that the set will have the same contents regardless of axiom application order.

Claim 4.1. *$PL(t)$ set value does not depend on calculation order and its calculation requires at most $Card(V'^*)$ applications of the predecessors graph axiom.*
Proof. Due to lack of space only a sketch of proof is given. $PL(t)$ calculation through application of axioms can be regarded as an iterative algorithm (see [3]) on G with a distributive function PL defined over a semilattice V'^* with operation of strict set inclusion. Since V'^* is a finite set the semilattice complies the finite chains condition (each chain contains at most $Card(V'^*)$ elements). Thus, we can apply theorem from [3], which states that $PL(t)$ calculation process is finite and does not depend on calculation order.
The PL set can be calculated by applying the following calculation algorithm:

```
$\forall t P(t) := P_e(t);$
$\forall t D(t) := D_e(t);$
$\forall t PL(t) := DL(t) := \{t\};$
while ( $\exists t \mid$ one of the axioms does not hold for $t$ )
   Recalculate $PL(t)$ applying axioms $4 - 7$ to $t$.
```

The `while` loop runs at most number of attributes by number of vertices times. The worst case graph for the algorithm is a bidirectional list with attributes in each vertex. ■

5 Elementary operations

Now when we have a set of invariants that establish requirements for G we consider a set of elementary operations of the evolution model. Each elementary operation should retain invariants defined in Section 4 if they were present in an initial schema.

We introduce eight elementary operations: adding/ removing an edge, adding/ removing an attribute, adding/ removing a vertex and splitting/ merging a vertex. Each of these operations has an equivalent operation on initial

DTD. We describe under what conditions the evolution model accepts an operation for a given schema, and what changes occur in the model when the operation is performed.

- `AddAttribute`. Attribute a (the one being added) of vertex t is included into $N_e(t)$. The sets N_e, N, H are recalculated for the vertex t and vertices reachable from t. Schema designer (or an external algorithm) may include this attribute into N_e sets of some successors of vertex t. The equivalent operation on simplified DTD is adding <!ELEMENT A> and including it in an element T: <!ELEMENT T(A,..)>.

- `RemoveAttribute`. Attribute a (the one being deleted) of vertex t is excluded from $N_e(t)$, the sets N_e, N, H are recalculated for the vertex t and vertices reachable from t. Note, if $t \in P(s) \wedge a \in Ne(s)$, then according to axioms 8 and 9 attribute a is included into $N(s)$. The equivalent operation on simplified DTD is removing element <!ELEMENT A> and excluding it from an element T: <!ELEMENT T(A,..)>.

- `AddEdge`. Let edge (t, s) is added. s is added into $P_e(t)$ and t is added into $D_e(s)$. Expressions dependent on $P_e(t)$ and $D_e(s)$ are recalculated. Note, that s is added into $P(t)$ if there is no other path from s to t (same for $D(s)$). The equivalent operation on simplified DTD is including element <!ELEMENT S> into element T: <!ELEMENT T(S,..)>.

- `RemoveEdge`. This operation is complex, it may cause generation of new edges in the schema according to axioms 4 and 6. New edges will start in some predecessors of the end vertex of a deleted edge and end in its successors. Provided we remove edge (t, s), s is deleted from $P_e(t)$. All expressions dependent on $P_e(t)$ are recalculated. If axiom 2 is violated then the operation is rejected by the system. (Alternatively, the vertex t may be included into root vertex T, the change can be achieved in two stages: adding T into $P_e(t)$, and RemoveEdge for the edge (s, t). Similarly, axiom 3 for vertex s may be violated. However, if $s \in P_e(t)$, then s is added into the graph). Analogous operations are performed with $D_e(s)$. In simplest case this operation equivalent is excluding element S from element T: <!ELEMENT T(S,..)> is replaced with <!ELEMENT T(...)>.

- `AddVertex`. Provided vertex t is added. t is included into V'. The sets $P_e(t)$ and $D_e(s)$ are to be defined by schema designer (or an external algorithm), to generate incoming edges of t. t is added into $P_e(s)$ to satisfy axiom 4. Basically, $P_e(t) = T$, and changes are made using modification AddAttribute. In case a new root is added the operation is equivalent to introducing new <!ELEMENT T(...)>, where ... contains an old root.

- `RemoveVertex`. It is a complex modification. First, the vertex t is to be removed with the edges starting and ending in it. Second, a number of edges from its predecessors to its successors may be added. Third, attributes of vertex t, that are essential for its successors should migrate properly. The operation is implemented in three stages: applying RemoveAttribute to all attributes of t, applying RemoveEdge to all outcoming edges, and applying RemoveEdge for all incoming edges. In the case of root removal the operation is equivalent to deleting a root element <!ELEMENT T(...)>, where ... contains a new root(s).

- `MergeVertex`. Vertices being merged must be connected by an edge. Merged vertex P_e, D_e and N_e sets are unions of P_e, D_e and N_e sets of vertices being merged with exclusion of themselves. In P_e sets of reachable amd D_e set of reaching vertices occurrences of merged vertices are replaced with occurrence of merged one. A merge equivalent is either replacing "*"-inclusion with ordinary one: <!ELEMENT T(S*)> is replaced with <!ELEMENT T(S)>, or removing inclusion of S into a third element V: <!ELEMENT V(..,S)> → <!ELEMENT V(...)>.

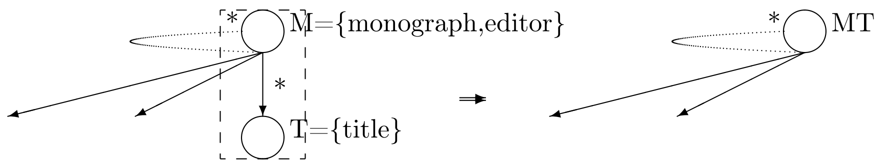

Figure 8. Merge vertex sample.

- `SplitVertex`. P_e, D_e and N_e sets of split vertex are separated into two disjoint sets each. In P_e and N_e sets of reachable vertices occurrences of split vertex are replaced with one or both of the created without violating $P(t) \subset P_e(t)$ and $D(t) \subset D_e(t)$ properties. A split equivalent is either replacing ordinary inclusion with a "*"-inclusion, or introducing inclusion of S into a third element V: <!ELEMENT V(...)> → <!ELEMENT V(..,S)>.

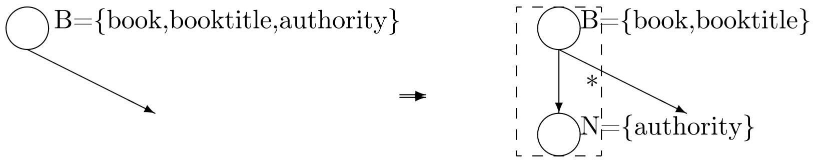

Figure 9. Split vertex sample.

The evolution from DTD99 to DTD01 for the working example may have the following sequence:

- MergeVertex(monograph,title)
- RemoveVertex(article)
  - RemoveAttribute(article,authorID)
  - RemoveAttribute(article)
  - RemoveEdge(article,author)
  - RemoveEdge(article,author)
- SplitVertex(book,authority,{authority},book)

- AddAttribute(authority,country)
- RemoveAttributeauthority
- AddAttribute(authority,authname)
- AddAAttribute(book,bestseller)
- AddAttribute(book,price)

The ability to apply one or the other elementary operation depends on the restrictions implied by the model axioms. In the evolution process the P_e, D_e and N_e (i. e. essential) sets of the model may need to be changed. The cost of an elementary operation is defined by the number of essential sets affected by it. The cost of application of a sequence of elementary operations is the sum of their costs. The preferred schema for a new version is one that requires less costly operation sequence to obtain it.

6 Conclusions

In this paper we presented a framework for schema evolution of relational database-based systems with XML "interface". Through definition of P_e, D_e and N_e sets of our model it becomes possible to ensure that evolved schemas designed to answer new functional requests while still conform to both, domain-based and performance-oriented restrictions.

Our future plans include designing a more flexible way to meet performance-oriented requirements than employing inline shredding technique. We also explore ways to extend the evolution model to control more XML DTD and XML-Schema features.

References

[1] R. J. Peters, M. T. Ozsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. ACM Transactions on Database Systems, 22(1), pp. 75–114, 1997.

[2] I. A. Goralwalla, D. Szafron, M. T. Szu. Managing Schema Evolution Using a Temporal Object Model. In proceedings of the 16 International Conference on Conceptual Modeling (ER'97), 1997.

[3] M. Yamamoto, K. Takahashi, M. Hagiya, S. Nishizaki, T. Tamai. Formalization of graph search algorithms and its applications. In proceedings of Theorem Proving in Higher Order Logics (TPHOLs'98), LNCS, vol. 1479, pp. 479–496. Springer-Verlag, 1998.

[4] S. Y. Lee, M.-L. Lee, T. W. Ling, L. A. Kalinichenko. Designing Good Semi-Structured Databases and Conceptual Modeling. In proceedings of International Conference on Conceptual Modeling / the Entity Relationship Approach, pp. 131–145, 1999.

[5] D. Florescu, D. Kossman. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report 3684, INRIA, 1999.

[6] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. J. DeWitt, J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In proceedings of 25th International Conference on Very Large Data Bases (VLDB'99), Edinburgh, Scotland, pp. 302–304, 1999.

[7] D. Florescu, D. Kossmann. Storing and Querying XML Data Using an RDBMS. IEEE Data Engineering Bulletin, 22(3), pp. 27–34, 1999.

[8] A. Deutsch, M. Fernandez, D. Suciu. Storing semistructured data with STORED. Proceedings of the 1999 ACM SIGMOD International conference on mamagement of data. pp. 431–442, 1999.

[9] Y. Men-hin, A. W.-C. Fu. From XML to Relational Database. In proceedings of 8th International Workshop on Knowledge Representation meets Databases, KRDB'2001, Rome, 2001.

[10] I. Tatarinov, Z. Ives, A. Y. Halevy, D. S. Weld. Updating XML. In SIGMOD, pp. 413–424, 2001.

[11] J. Shanmugasundaram, E. J. Shekita, J. Kiernan, R. Krishnamurthy, S. Vigilas, J. F. Naughton, I. Tatarinov. A General Techniques for Querying XML Documents using a Relational Database System. SIGMOD Record, vol. 30(3), pp. 20–26, 2001.

[12] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. International. In proceedings of the 2002 ACM SIGMOD international conference on Management of data table of contents, Madison, Wisconsin, pp. 204–215, 2002.

[13] B. Kane, H. Su, E. A. Rundensteiner. Consistently updating XML documents using incremental constraint query checks. In Web Information and Data Management (WIDM'02), pp. 1–8, 2002.

[14] S. Coox. Axiomatization of Schema Evolution in XML Databases. In Programming (3), pp.1-9, 2003.

[15] Y. Papakomstantinou, V. Vianu. Incremental validation of XML documents. In ICDT, 2003.