Deductive Approach to Semistructured Schema Evolution *

© D. Luciv Ph.D. adviser: B. Novikov

Saint-Petersburg State University
dluciv@lanit-tercom.com

Abstract

Schema evolution for different methodologies is an interesting area of research for now and future.

Some results of research of semistructured database schema evolution are presented here. Semistructured database schema model based on [11, 4] and its evolution techniques are introduced. Extensions to data model are also described.

Some refactoring problems are described. Declarative approach to schema evolution and refactoring is presented.

1 Introduction

As one of the most popular universal data interchange formats, XML [2] needs to be structured and constraints applied to XML data are to be formalized.

XML is now often used for short message interchange and storing small pieces of data for particular applications. Number of such applications is relatively big and in this case data structure of XML can be restricted with only constraints applied by XML itself: the data should have hierarchical form with tags and attributed and should obey XML syntax.

For more serious applications XML data schema constraints are practically defined and then checked using such means as *DTD* [12], *XML Schema* [21], *Relax-NG* [1], etc. Although for productivity reasons genuine-XML (unparsed textual) data packages are still relatively small for such applications, they can have a complicated structure.

For all methodologies maintaining data structure is interesting, but difficult. One of research areas is database schema evolution for relational, object-oriented [11], semistructured hierarchical [4, 5, 10, 14, 15] databases, and a number of related areas like XML-relational mapping [3, 7, 8, 14].

Traditionally, data schema evolution is not a part of the both object-oriented, relational and semistructured data definition methodologies, however data structure often requires being alterable. This comes from practice. As an illustration we can say that almost any relational database has powerful interface for schema altering, and relational databases are mostly used to store large amounts of data. Other kinds of databases often use relational engines behind their semistructured and/or object-oriented interfaces to increase productivity.

Usually people think about real-world concepts like entities, relations and attributes when modeling the data. The same can be said about data schema evolution controlled by human operator. Existing semistructured database schema evolution techniques provide "low-level" evolution operations over particular schema items like tags, attributes, associations, etc. However sometimes operator thinks about more high-level operations like splitting entities, moving attributes from one entity to other, etc - which can be called "refactoring".

This paper describes some investigations in area of schema evolution and refactoring and possible approaches to it.

The remaining paper is organized as follows:

1. related work is analyzed;
2. basics for semistructured hierarchical data modeling are defined;
3. restrictions on XPath expressions are defined and applied when modeling associations;
4. schema invariants are described;
5. declarative approach to invariants implementation is presented;
6. declarative approach to refactoring is introduced, refactoring operations are described;
7. some conclusions are presented.

2 Related Work

Schema evolution is an interesting problem for almost all data modeling methodologies.

Object-oriented database schema evolution technique for TIGUKAT ODBMS was proposed in [11]. This model was selected as basic one to describe schema evolution for hierarchical semistructured databases, particularly, XML in [4].

Both [11] and [4] models are simple and illustrative. To make them more flexible, they were modified to handle regular expressions [5] and reference-based associations [10].

We already have results in extending existing semistructured database model by adding associations to it. Like [5], we base on [4] model for document tag structure. New rules for association evolution are also added.

* This research was partially supported by Microsoft Research grant No. 2004-459A and RFBR grant No. 04-01-00173

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems SYRCoDIS, St.-Petersburg, Russia, 2005

Another interesting research area of research is onthology evolution [18, 17]. Some onthology models, like conceptual graphs [16], can be considered a conservative extension to first-order logic and used to express XML data model too. Example Prolog interpreter with conceptual graph support can be obtained at `http://prologpluscg.sourceforge.net/` as a part of Amine Platform (`http://amine-platform.sourceforge.net/`).

3 Evolution of Data Schema

3.1 Data Modeling Basics

Let us consider sample data model to be managed (fig. 1).

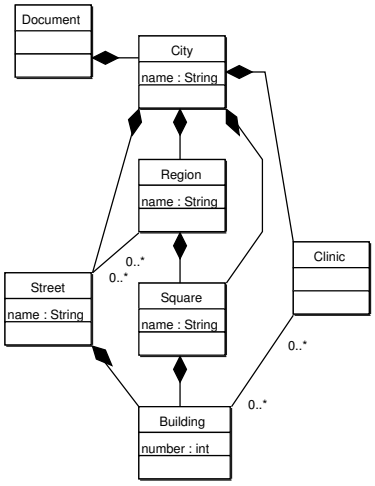

Figure 1: Data Structure

Given data structure mostly consists of entities and aggregations between them. Some associations are not supported by XML syntax: in fact XML supports only *strong aggregation*. Other kind of associations that is described in [10] and in this paper are associations based on XML references. Those associations are one-way navigable.

The model needs some change to become realistic. Modified model is shown on fig. 2. Here we have resolving entities for **Street** and **Building**. Two resolving tags for entities those are to be referenced using "$? \rightarrow *$" associations are here added to the model. All associations are now references with single direction (as navigability shows). By the way, resulting second model is poorer than first one: for example, we can't list **Region**s containing given **Street** using revised data model.

3.2 Notations and Definitions

3.2.1 Basic Notations

Following notations are defined by [4]:

τ	tag graph
s, t	tags
a	attribute
m	tag *or* attribute
$N(t)$	"native" attributes of t
$N_e(t)$	essential "native" attributes of t

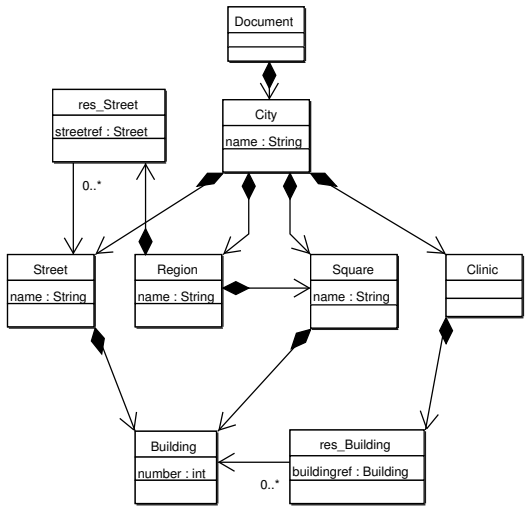

Figure 2: Real (physical) Data Structure

$P(t)$	set of immediate predecessors of tag t
$P_e(t)$	set of essential immediate predecessors of tag t
$PL(t)$	t predecessors graph
$H(t)$	t inherited attributes
$I(t)$	t interface - inherited and native attributes
$\mathsf{inst}(s), \mathsf{inst}(a)$	instances of tags s or a
$\mathsf{val}(\mathsf{inst}(s))$, $\mathsf{val}(\mathsf{inst}(a))$	values of $\mathsf{inst}(s), \mathsf{inst}(a)$ respectively
A	union of all tags' native attributes

$\alpha_x(f, \tau) = \{f(x) | x \in \tau\}$ – the set of values of f where x passes τ

Other definitions, dedicated for association modeling, are introduced below.

All desired invariants (schema axioms) are defined in [4, 10].

3.2.2 Restrictions To XPath, Association Modeling

Here we constrain XPath expression class to be simple enough to analyze. Some of described notions were already defined and used in [10], here they are described broader.

Let us consider association to be reference from tag t towards tag s. Tag s is identified by b attribute, tag t uses a attribute to refer tag s. The reference is implemented using XML *key* and *keyref* constructs.

Example XPath expression looks like
" `max(/Document/City/Street/Building [parent::Street/@name="Smiths'"]/@number)`".

It contains attributes (`@name, @number`), axes (implicit `child::` and explicit `parent::`), predicate (`...="Smiths'"`) and term (`max(...)`). See[19] for more information.

Although above XPath expression does not employ every possible XPath functionality (e.g. "|", other operations and advanced features), it is complicated enough. Here we do not need to formalize every kind of XPath expression.

The biggest own subclass of XPath expressions defined in this paper is a class of *Simplified* expressions. Such expressions do not use axes (except `child::`), predicates, terms and operations. The most powerful element of this class is "//" axis. This axis can be easily simulated by substitution of "//" with all possible path sections and then uniting search results. This operation is finite because [4] does not use cycles in document schemas.

Simplified expressions, like other XPath expressions, are in real XML documents, not schemas. However *Simplified* expression can be described using document model terminology. It can be considered as a function $p : \tau \to 2^{\tau}$ because if we know what tag is an argument instance of, we always know all possible tags which are resulting values instances of.

We can constrain result set of *Simplified* expression this way: $\sharp\{p(t)\} = 1, \forall p \in SP,\ t \in \tau$.

Members of such expressions class act as $SP \ni sp : \tau \to \tau$. SP class is called *Simple* expressions here. We here denote the set of simple XPath expressions as SP, so $SP(t)$ is the set of tags s which can be reached from t using simple XPath expressions. So $\alpha_t(SP(t), \upsilon) \subset \tau$ for $\upsilon \subset \tau$. We are also to define A as $\bigcup_{t\in\tau} N(t)$. So notation now includes:

$SP(t)$	subgraph of τ reached by simple XPath expressions from t
A	union of all tags' native attributes

Above explanations' goal is to make nature of SP easy to understand. Formally we can define SP as $\{s|t \in PL(s)\} \cup t$.

Associations are represented with keys and references [10]:

$$\mathsf{keydecl} : \tau \times SP \times (A \cup \tau) \to K, \text{ so}$$
$$K = \{f_k : \{\mathsf{val}(m)\} \to \{\mathsf{inst}(s \in \tau)\}\},$$
$$f_k \text{ is the } \textit{key function}$$

and

$$\mathsf{keyrefdecl} : \tau \times SP \times (A \cup \tau) \times K \to R, \text{ so}$$
$$R = \{f_r : \{\mathsf{inst}(t)\} \to \{\mathsf{val}(m)\}\},$$
$$f_r \text{ is the } \textit{key reference function}.$$

Finally, for instances of tags we require

$$f_k = \mathsf{keydecl}(t_1, p_1, m_1), f_r = \mathsf{keyrefdecl}(t_2, p_2, m_2, f_k)$$
$$t_1 \in PL(t_2);$$
$$\forall e_2 \in \{\mathsf{inst}(p_2(t_2))\} \quad \exists e_1 \in \{\mathsf{inst}(p_1(t_1))\} :$$
$$(f_k \cdot f_r)(e_2) = e_1.$$

3.2.3 Sets

Following notations for key and reference modeling are introduced by [10]:

$KC(f_k)$	Coverage of key with function $f_k = \mathsf{keydecl}(t, p_1, a_1)$
$RC(f_r)$	Coverage of reference with function $f_r = \mathsf{keyrefdecl}(s, p_2, a_2, f_{k'})$
$CK(t)$	Key declarations covering $t \in \tau$
$CR(t)$	Reference declarations covering $t \in \tau$
$NK(t)$	"Native" key declarations of $t \in \tau$
$NR(t)$	"Native" reference declarations of $t \in \tau$
$CR_e(t)$	"Essential" reference declarations of $t \in \tau$. Here we mean reference declarations which base on essential attributes of referencing tag and it's essential parents
$NR_e(t)$	"Essential native" reference declarations of $t \in \tau$. Here we mean reference declarations which base on essential attributes of referencing tag

3.2.4 Invariants

The set of axioms (denoted as (1)–(9) here) given in [4] is extended by following ones in [10]:

Axiom	Formal representation
(KR1)	$NK(t) = alpha_s(\mathsf{keydecl}($ $s, p : p(s) = t, a \in N(t)), PL(t))$
(KR2)	$NR(t) = \alpha_s(\mathsf{keyrefdecl}($ $s, p : p(s) = t, a \in N(t), *), PL(t))$
(KR3)	$\forall f_r \in R \ \ \exists f_k \in K :$ $f_r = \mathsf{keyrefdecl}(t \in \tau, p \in SP,$ $m \in (A \cup \tau), f_k)$
(KR4)	$\forall t \in \tau \ \ CR(t) \in \alpha_s($ $\mathsf{keyrefdecl}(*, *, *, s), CK(t))$
(KR5)	$NR_e(t) = \alpha_s(\mathsf{keyrefdecl}($ $s, p : p(s) = t, a \in N_e(t)), PL(t), *)$

They are named as axioms of -native keys, -native references, -declaration, -coverage, -"essential" references respectively.

Consistency and completeness of invariants (1)–(9) are proved in [4] by following theorem:

Theorem 1 *Schema axioms are sound and complete, so if we have $P_e(t)$ and $N_e(t)$ defined for each tag in schema our schema satisfies axiom set and making changes to this schema is govered by the sets.*

Proof

is given in[4]

End of Proof

Key and reference invariants and formulated in [10] and proof idea can be expressed as follows:

Theorem 2 *Schema axioms are sound and complete, so if we have $P_e(t)$, $N_e(t)$, $NK(t)$, $NR_e(t)$ defined for each tag in schema our schema satisfies axiom set and making changes to this schema is governed by the sets.*

Proof

Theorem 1 says that schema without key and reference declarations satisfies axioms (1)–(9) when $P_e(t)$ and $N_e(t)$ are defined for each tag. But axioms (1)–(9) say nothing about keys and references so new model satisfies those axioms. The same fact relates to model without keys and references and axioms (KR1)–(KR5): no proof needed.

Hence we are now to prove that keys and references of document schema satisfy new axioms (KR1)–(KR5).

(KR1),(KR2) Those axioms are satisfied obviously due to key and reference semantics presented here.

(KR3) Every reference should base on corresponding key.

(KR4) We have noted above that reference coverage is always contained by key coverage. Thus if we use reference covering our tag, we are also to have key declaration covering our tag.

(KR5) Like (KR1),(KR2) is satisfied due to reference semantics. It is always able to build $NR(t)$ set using $NR_e(t)$ and $N(t)$ sets.

End of Proof

4 Declarative Approach

4.1 Description of Schema

All schema invariants are already defined using first-order logic. Evolution is done by modifying governing sets [4] and recalculating other ones.

Possible way to make the schema practically declarative is to assume XML document schema stored in deductive database.

Description below is equivalent to above schema invariants but can be programmed using Prolog [9] in nearly unchanged form. However description is simplified here to make it more illustrative yet functional.

Note that next section describes data schema facts, but does not describe any method to ensure schema is consistent. It is not a vulnerability of the approach: evolution operation set is full, any operation leaves correct schema in correct state, so any correct schema can be "unrolled" from empty one.

Example set of facts below can be interpreted as state of schema at particular moment. Loading data schema state "as-is" can also be useful, like restoring relational databases from dumps with constraints disabled temporarily for productivity.

4.1.1 Facts of Schema

Facts of deductive databases are used to store information that can not be deduced. This is primary information about data schema.

Illustration below defines subset of entities from data model shown on fig. 2. Schema can be described as follows:

```
%--- EntityModel
tag(':root'). tag(':term').
tag('Document'). tag('City').
tag('Square'). tag('Street').
tag('Region'). tag('res_Street').
inPe('City', 'Document').
inPe('Square', 'City'). inPe('Square', 'Region').
inPe('Street', 'City'). inPe('Region', 'City').
inPe('res_Street', 'Region'). % - non-essential
inPe(':term', 'Square'). inPe(':term', 'Street').
```

Notes (here and below):

- Prolog syntax is somewhat changed to optimize paper usage and to group some data;
- attributes are named uniquely for simplicity to distinguish them by name.

In case we can use model from fig. 1 directly (our methodology allowed them), we should write a fact like `inPe('Street','Region')`. Moreover, Schema defined is not complete in area of `Region`→`Street` reference because `res_Street` references `Street` via reference based on attribute `streetref`.

Attributes can be described as follows:

```
%--- AttribModel
inNe('City', 'City@name').
inNe('Street', 'Street@name').
inNe('Region', 'Region@name').
inNe('Square', 'Square@name').
inNe('res_Street', 'res_Street@streetref').
```

Two sections of Prolog facts above define almost everything needed to describe model as [4] proposes, but references defined in [10] are also to be defined in case when we consider them as evolution object.

keydecl and keyrefdecl constructs require a bit more explanation before defining them. Reference axiomatic in [10] is based on key and reference functions which allow us, when superposed, identify referred tag instance for referring one including all references defined for it.

Those functions are "generated" by keydecl and keyrefdecl which are, formally, functionals. Such a description with functionals and functions can be left almost unchanged when implementing XML DBMS using a kind of functional language for it, but we use logical language to describe data schema here.

keydecl construct can be used to identify resulting key function keydecl(t, p, a) as:

```
%--- RefersModel
%---- Keys
inNK('City', ['Street'], 'Street@name').
```

or, with other context:

```
inNK('Document', ['City', 'Street'],
'Street@name').
```

in case we want document-unique street name.

It is possible to reach key-covered tag by different ways so we need a list (or other ordered container) to represent path from SP here.

keyrefdecl functional gets keydecl result as an argument, but it is correct to say that keydecl can be identified by it's argument, so practically keyrefdecl gets 6 parameters in deductive database, e.g.:

```
%--- RefersModel
%---- Refs
inNRe(
 'City', ['Region', 'res_Street'],
 'res_Street@streetref',  'City',
 ['Street'], 'Street@name' ).
```

4.1.2 Goals of Schema

Sets which are not yet defined using Prolog are *calculable* and can be omitted theoretically, however [11, 4, 10] defined them for simplicity, and we will also use those definitions below.

In deductive database those sets should not be asserted during schema evolution, they can be considered as "read-only" and calculated using schema invariants. Thus they are defined using goals, not facts.

```
%--- Calculable Sets

 % Pred. Graph
 inPL(S, T) :-
  inP(S, T).
 inPL(S, T) :-
  inP(S, U),
  U=\=T,
  inPL(U, T).
 inPL(S,S).

 % Immediate Pred.
 inP(T, S) :-
  inPe(T, S),
  inPe(T, X),
  X=\=S,
  not(inPL(X,S)).
 inP(T, S) :-
  inPe(T, S),
  tag(X),
  X=\=S, %single Pe
  not(inPe(T,X)).

 % Inh. attributes
 inH(T, A) :-
  inP(T, S),
  inI(S, A).

 % Native Attributes
 inN(T, A) :-
  inNe(T, A),
  not(inH(T, A)).

 % Interface
 inI(T, A) :-
  inN(T, A).
 inI(T, A) :-
  inH(T, A).

 % :root & term
 inPL(_,':root').
 inPL(':term',_).
```

Above Prolog goals can be used almost without changes except recursive goals for PL sets. In experiments they were optimized and allowed to check if any of arguments are not free term and then decide how to recur. For example, when we try to solve `inPL(X, 'City')` it is better to recur down from `City` tag, however axioms and goals above recur upwards only. Calculation of P sets was also changed in practice: in [11, 4] invariants were defined to handle sets of unique elements, however Prolog gives a number of non-unique solutions for `inP` and `inPL` above goals (all variants) for any unbound arguments. Practically, some tricks with Prolog cut-off were used. All those tricks and optimizations are beyond this paper.

Detailed descriptions of sets above and formulae for schema invariants are placed in [4]. Those formulae are used "as-is" their calculation require no imperative description at all.

$NK(\tau)$ and $NR_e(\tau)$ sets are governing sets for keys and references, other are calculated. Invariants from [10] can be described like this:

```
last(l, e) :-
 eq(l, (h)),  eq(e, h).
last(l,e) :-
 eq(l, (h|t)), last(t,e).

inNR(Rtag, Rpath, Rattr, Ktag, Kpath, Kattr):-
 inNRe(Rtag, Rpath, Rattr, Ktag, Kpath, Kattr),
 last(Rpath, RRtag),
 inN(RRtag, Rattr).
```

4.2 Declarative Approach to Evolution

Basic operations which were initially defined in [4] for the data model without associations are modified to handle references. Evolution operations from [10] are here defined considering schema as deductive database described above.

Note that we use some ISO Prolog goals like `findall(Var, Term, Result)` [6]. However Prolog code below can be used as-is, this code is dedicated to illustrate main ideas of possible schema evolution engine implementation. Some checks are here omitted and evolution operations are defined as simply as possible.

Evolution operations are labeled «***evo.M.N***» below.

evo.1.1. adding new attribute

Rule described in [4]. No additional requirements and activities needed.

```
addAttribute(Tag,Attr) :-
    tag(Tag),
    assert(inNe(Tag,Attr)).
```

evo.1.2. deleting attribute

When deleting attribute a of tag t, all key declarations in $NK(t)$ and references in $NR(t)$ are deleted. See operation of key removal (3.2) below.

As noted in [4], an ability to add deleted attribute to N_e of tags in $SP(t)$ is useful. Implementation of this functionality should also *redeclare* keys in $NK(t)$ and references in $NR(t)$. Sample implementation is cumbersome and omitted here.

evo.2.1. adding aggregation between two tags

Rule described in [4]. No additional requirements and activities needed.

```
addAggregation(Tag,Parent) :-
    tag(Tag), tag(Parent),
    assert(inPe(Tag,Parent)).
```

evo.2.2. removing aggregation between two tags

This operation is relatively complicated in both [4] and here.

Let us call aggregated tag t.

All key declarations in $NK(t)$ are modified to satisfy (KR4) axiom. This means that declarations move upward unless they cover all references again. Note that if t is no more aggregated by other tags it implicitly becomes child of root tag $\top$, and all corresponding key declarations move to $\top$.

All reference declarations in $NR(t)$ are deleted if they become invalid. All reference declarations in t move up to save references in $\alpha_s(NR(s), SP(t))$.

A sample code (qualitatively simplified) here executes key redeclaration according to (KR4).

```
leastCommonEPL(Tag1,Tag2,Pred,Path1,Path2) :-
...
% finds least common predecessor in both
% PL(Tag1)\{Tag1} and PL(Tag2)\{Tag2} and
% pathes to it. Trivial but cumbersome.

retractDependentRefs(Child, Parent, DKP) :-
 findall((RT,RP,RA,KT,KP,KA),
 (inNRe(RT,RP,RA,KT,KP,KA),
  member(Child, KP),member(Parent, [KT | KP]),
  retract(inNRe(RT,RP,RA,KT,KP,KA)) ), DKP).

correctKeyRefs(Child, Parent, DK) :-
 findall(inNRe(RT,RP,RA,KT,KP,KA),
 (inNRe(RT,RP,RA,KT,KP,KA),
  member(Child, KP),
  member(Parent, [KT | KP]),
  retract(inNRe(RT,RP,RA,KT,KP,KA)),
  retract(inNR(KT,KP,KA)),
  leastCommonEPL(Parent,Child,LKE,PPath,CPath),
  addKey(LKE,CPath,KA),
  addReference(RT,RP,RA,LKE,PPath,KA)
 ), DK).
```

Then we can invoke them:

```
removeAggregation(Child,Parent) :-
 retractDependentRefs(Child, Parent, _),
 correctKeyRefs(Child, Parent, _),
 retract(inPe(Child,Parent)).
```

evo.2.3. adding new tag

The rule is described in [4]. No additional requirements and activities needed. Prolog goal below also illustrates some checks of bound variables. Those checks are useful live applications.

```
addTag(Tag, PeList) :-
 findall(X, (member(X, PeList),
  not(tag(X))), BadList), %only sample check
 length(BadList, 0), % they all are tags
 assert(tag(Tag)),
 findall(X, (member(X, PeList),
  addAggregation(Tag, X)), Successfull),
 addAggregation(':term', Tag).
```

evo.2.4. removing a tag

The rule is described in [4] for basic structure without keys and references. Removed tag is called t. Like in operation (1.2) of attribute removal, the reference declarations in $NR_e(t)$ can be modified to cover all children of t. For all such reference declarations we are to create corresponding reference attributes in children of t. Sample Prolog implementation is not placed here because of its awkwardness.

evo.3.1. declaring new key

Key declaration is canceled if it does not satisfy one or more of (KR1)-(KR5). Sample Prolog code omits checks because they are cumbersome and can be considered as unnecessary technical details here. It requires almost no checks in theory, either.

```
addKey(ContextTag, Path, KeyAttr) :-
 % no checks
 assert(inNK(ContextTag, Path, KeyAttr)).
```

evo.3.2. removing a key

During key declaration $f_k = \mathsf{keydecl}(t, p, a)$ removal, all reference declarations $f_r = \mathsf{keyrefdecl}(t', p', a', f_k)$ are also removed. Sample Prolog implementation is not interesting enough to place here.

With current key-reference model is is the only way to perform such an operation. However if we support *Simplified* XPath expressions instead of *Simple* ones, we can identify instances of many tags with one key. In this case we potentially can redeclare the key by adding it to NK sets of t tag's children. This will keep references to f_k unchanged. But this case is not considered here.

evo.4.1. declaring new reference

Reference declaration is canceled if it does not satisfy one or more of (KR1)-(KR5). Sample Prolog implementation is not interesting enough to place it here.

evo.4.2. removing reference

This operation does not require any additional operations. Like operation (1.2), it can be compensated with declaration of references belonging to NR sets of some child tags. Sample Prolog implementation is not interesting enough to place here.

In case when we support *simplified* XPath expressions, we can cover some child tags with one reference declaration instead of declaring new references, but we do not consider this way here as it was noted for (3.2) operation.

4.3 Declarative Approach to Refactoring

Human-language description of evolution operations is cumbersome and complicated. However this is only a set of possible low-level schema modification operations. Refactoring operations are rather more complicated than ones above. Imperative description of refactoring operations should be even more complicated and unclear so defining them in declarative way seems to be a way to simplify them.

Here some possible refactoring operations are listed and sample implementation ideas provided. Wider operation list can be found at `http://www.agiledata.org/` in Database Refactoring Catalog section.

Refactoring operations listed below are mostly projected into sequences of schema evolution operations. Some of refactoring operations require analysis of real data corresponding to given schema. Some of them are unapplicable yet because XML Schema support in data model above is not yet powerful enough.

4.3.1 Refactoring Operations over Existing Data Model

Refactoring operations defined and described below are labeled «***rft.N***» below.

rft.1. consolidating attributes representing single concept

For attributes, which represent single concept, consolidation can be executed. For example, address is sometimes represented by two strings and can be consolidated into one string.

rft.2. consolidating tags

Sometimes two model tags can be consolidated because they can be interpreted as the same entity.

rft.3. splitting an attribute

Splitting an attribute is useful when the model is growing from more simple one or from a prototype. Example above can be reversed: if we know particular addressing system, we can divide attribute into smaller ones representing more atomic values.

rft.4. splitting a tag

Splitting a tag can be useful, for example, when we would like to detach a child tag from parent (instead of adding attributes to all instances of existing tag).

rft.5. adding a key to data schema

Adding new key operation is basic evolution operation and it is already defined. Although in real databases adding a key to schema can be also combined with integrity checking and index creation. Automating of those tasks can lead to complicated refactoring operation.

rft.6. adding new attribute to avoid referencing

Instead of dereferencing other entity to obtain the value of single attribute, we can store this attribute in the referencing entity. Although productivity can be essentially improved, this approach can lead us to inconsistent data.

rft.7. moving an attribute

Moving from one tag to another lets us to:

- avoid possible referencing when accessing attributes of different entities;
- separate seldomly-used attributes to improve productivity;
- separate read-only attributes if DBMS has no permission support below entity level or when it helps DBMS to optimize its work.

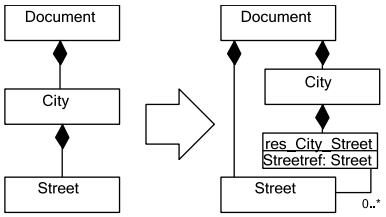

Figure 3: Refactoring Model fragment

rft.8. moving a tag within τ

This operation is very common and affects the whole schema. It should be decomposed into sequence of aggregation removal and creation.

rft.9. safe-deleting entity

This refactoring operation is already supported as elementary evolution operation.

rft.10. replacing existing natural key with surrogate one

This operation declares new key and probably removes existing one (existing key becomes simply an attribute). It can also try to maintain references to initial key and switch them to new one.

The purpose of this operation is to change key semantics.

rft.11. replacing "$1 \to *$" aggregative association with "$* \to *$" one

As an example we can imagine that we are going to let the streets to pass across several cities (fig. 2). In this case `Street` becomes a child of `Document` and new resolver between `City` and `Street` is constructed as shown at fig. 3.

For this refactoring operation sample implementation is provided. It is quite simple (some supplementary goals are also defined, obvious goals are used without definition):

```
useOrCreateKey(Tag, PathWithoutLast,
 KeyTag, KeyAttr, MustReferTag):-
 concat(PathWithoutLast, [KeyTag], Path),
 inNK(Tag, Path, KeyAttr),
 inPL(MustReferTag, Tag),
 !. %one is enough
useOrCreateKey(Tag, PathWithoutLast,
 KeyTag, KeyAttr, MustReferTag):-
 leastCommonEPL(KeyTag,MustReferTag,
  Tag,Path,RPath),
 KeyAttr = Keytag + '@idFor_' + MustReferTag,
 addAttribute(KeyTag, KeyAttr),
 addKey(Tag, Path, KeyAttr),
 removeLast(Path, PathWithoutLast).
replaceAggregationWithResolver
 (Child, Parent, ResName) :-
 ResName = 'res_' + Parent + '_' + Child,
 addTag(ResName, [Parent]),
 removeAggergation(Child, Parent),
 RefAttrName = ResName + '@' + Child + 'ref',
 addAttribute(ResName, RefAttrName),
 useOrCreateKey(Tag, PathWithoutLast,
  Child, KeyAttr, ResName),
 concat(PathWithoutLast, [KeyTag], Path),
 onePathFromTo(ResName, Tag, ResPath),!,
 %gives one, cuts off.
 addReference(
  Tag, ResPath, RefAttrName,
  Tag, Path, KeyAttr
 ).
```

4.3.2 Refactoring Operations over Advanced XML Data Structures and Models

rft.12. renaming anything

This refactoring operation is placed in "advanced" section because current data model does not handle such advanced XML naming facilities as namespaces, etc. If these facilities are appropriately supported, renaming tag, attribute, key or any other entity in XML data model becomes complicated task.

rft.13. restore standard types for attributes

Theoretically, this operation is very interesting. Refactoring engine should analyze user types defined in data schema, perform some calculations of corresponding domains and decide if some of types can be replaced by built-in (like integer) types. Moreover, merging similar types (and type fragments of compound types) and simplifying type description is very interesting computer science problem. This interesting operation is beyond current work.

rft.14. adding cascade delete for weak entities

This operation creates implicit triggers in underlying DBMS in case one exists to delete weak entities when their parents are destroyed.

rft.15. discover groups and turn them into entities

Existing data can be analyzed to discover repeating values of attributes and tags. Some of them can be merged and then replaced with references to newly created merged groups.

5 Conclusion

Given association evolution model makes a contribution to existing XML schema evolution models. Although given model does not provide full support of XML Schema or even DTD constructions, it seems to be enough powerful for usage with real XML documents.

Model can be modified if needed. For example, support of simplified XPath expressions can make model much more flexible. New association model can be also integrated into document model described in [5] instead of [4]. Both solutions increase model flexibility and DTD or XML schema support, but resulting model will be still not enough powerful to support complete DTD and XML schema. Both solutions lead to complicated model that is not so illustrative as given one, so they are to be used in cases when their techniques are useful for particular applications.

It can be sensible to combine some other set calculation models together with Prolog goal approvals. Sample sample system is term rewriting engine [13] available at `http://www.gradsoft.com.ua/`.

Although above paper describes schema evolutions, one of possible applications of given evolution rules is automatic transformation of XML documents content. For example, two versions of database can have different XML representation of their data. On-the-fly transformations of XML content can help legacy applications to access new database and can also make legacy database available for new applications. Transformation itself can be composed using operations described in this paper. For real world application this transformation can be executed by XSLT [20] generated from evolution operations list.

For XML document with complicated structure it is easier for human operator to think about more "high-level" transformations, than most of primitive evolution operations defined in [4, 5, 10] and here. Such operations, like splitting entity or moving attribute from one entity to another, etc. should be considered schema refactoring, because in their case simple local changes often induce global changes to whole schema. Some of possible schema refactoring operations are described here and are also objects of future investigations.

References

[1] Relax ng compact syntax, November 2002. Specification.

[2] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds). Extensible markup language (XML) 1.0 (2nd edition). W3C Recommendation, 2000.

[3] Surajit Chaudhuri, Raghav Kaushik, and Jeffrey F. Naughton. On relational support for XML publishing: Beyond sorting and tagging.

[4] S. Coox. Xml database schema evolution axiomatization. *Programming and Computer Software*, 29(3):140–146, 2003.

[5] Sergey V. Coox and Andrey A. Simanovsky. Regular expressions in xml schema evolution. Kharkiv, Ukraine, June 2003. ISTA.

[6] DEIS, Universit'a di Bologna a Cesena, Italy. *tuProlog User's Guide*, 1st edition, Sep 2002.

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

[8] D. Florescu and D. Kossmann. Storing and querying xml data using an rdbms. In *Data Engineering Bulletin*, volume 22, pages 27–34. IEEE, 1999.

[9] J.Doores, A.R.Reiblein, and S.Vadera. *Prolog – Programming for Tomorrow*. Sigma Press, Wilmslow, UK, 1987.

[10] Dmitry V. Luciv. Semistructured database scheme evolution and refactoring. In *SYRCoDIS*, pages 71–74, May 2004.

[11] M. Tamer Oszu Randal J. Peters. Axiomatization of dynamic schema evolution in objectbases. 1997.

[12] Erik T. Ray. *Learning XML.* O'Reilly, 2001. pp. 143-189.

[13] R. Shevchenko and A. Doroshenko. The system of symbolic computing for programming the dynamic applications. *Interntet publication*, 2003. (in russian).

[14] A. Simanovsky. Evolution of schema of xml-documents stored in a relational database. In *proceedings of Baltic DB&IS*, Riga, Latvia, 2004. Association for Computing Machinery. To appear.

[15] Andrew A. Simanovsky. Applying the reconfiguration-design formalism to xml stored in a relational database. In *SYRCoDIS*, pages 75–77, May 2004.

[16] John F. Sowa. Conceptual graphs. ISO/JTC1/SC 32/WG2, April 2001. Standard Draft.

[17] L. Stojanovic, A. Maedche, N. Stojanovic, and R. Studer. Ontology evolution as reconfiguration-design problem solving. In *proceedings of International Conference on Knowledge Management*, 2003.

[18] M. Stumptner and F. Wotawa. Model-based reconfiguration. In *proceedings Artificial Intelligence in Design, Lisbon, Portugal.*, 1998.

[19] W3C. *XML Path Language (XPath)*, 1.0 edition, November 1999.

[20] W3C. *XSL Transformations (XSLT)*, 1.0 edition, November 1999. Recommendation.

[21] W3C. *XML Schema Part 0: Primer*, May 2001. Recommendation.