XML schema mappings using schema constraints and Skolem functions *

Tadeusz Pankowski[1,2]

[1] Institute of Control and Information Engineering,
Poznań University of Technology, Poland
[2] Faculty of Mathematics and Computer Science,
Adam Mickiewicz University, Poznań, Poland
email: tadeusz.pankowski@put.poznan.pl

Summary. A schema mapping is an executable specification describing transformation of data structured under different schemas. In this paper we discuss the problem of automatic generation of XML schema mappings using information provided by schemas and correspondences between schemas. Mappings are specified in a mapping language XDMap whose constructs are based on Skolem functions. We use Skolem functions with text-valued arguments from a source instance to create nodes in the target instance, and to specify functional dependencies between some values. First, using constraints defined within a schema, an algorithm produces the automapping representing this schema. Next, (auto)mappings can be combined by means of some operators delivering more general mappings between schemas provided that a correspondence between schemas is given. While a mapping is executed, some missing data can be inferred based on constraints encoded in the mapping specification.

1.1 Introduction

A schema mapping is a specification that describes how data structured under a source schema is to be transformed into data structured under a target schema [8]. Recently, this problem has received considerable attention in the context of data exchange [3, 7], data integration [12, 18], schema evolution [14], P2P databases [5, 19], life science databases [22] or e-commerce [6, 21], where data comes from many different sources with different schemas.

In this paper we are continuing our work on automatic generation of mappings and transformations of XML data [17, 22, 6, 19, 20]. The proposed method is based on *constraints* defined within XML schema: *keys*, *key references*, and *value dependencies*. We show how these constraints can be formalized and how they can be used to generate *automappings*. An automapping

* The work was supported in part by the Polish Ministry of Science and Higher Education under Grant N516 015 31/1553.

maps a schema onto itself. Automappings can be next combined giving arbitrary mappings between different schemas by means of *Match*, *Compose*, and *Merge* operators over (auto)mappings [19]. In order to express unique relationships between data we use Skolem functions in specification of mappings in XDMap. We also show how some missing or incomplete data, which are not given explicitly in sources, can be deduced based on value dependency constraints while a mapping is executed.

To define mappings we assume that key and some value constraints are specified within schema (using XML Schema [27] notation). We show how an automapping may be automatically generated from these constraints. It is significant in our approach that the constraints are specified outside the mapping by means of constraint-oriented notation. The generated automapping preserves these constraints. In contrast, in other mapping languages (e.g. in [28]) constraints must be explicitly encoded in the mapping language. This can make difficulties for future management when schemas evolve.

It is commonly accepted that the basic relationships between a source and a target relational schemas can be expressed as a source-to-target dependencies (*STD*) [2, 9, 14, 15]. In [3] *STDs* are adopted to XML data in such a way that if a certain pattern occurs in the source, another pattern has to occur in the target. In our approach, the main idea of using *STDs* consists in specifying how nodes in a target instance depend on key paths, how these key paths correspond to paths in sources, and how target values depend on other values. So, our approach is more operational and uses DOM interpretation of XML documents. To generate the instance of a target schema from instances of source schemas, we use the idea of *chasing* [2, 28].

The paper is organized as follows. In the following section we discuss and propose some definition formalizing notions which are used in the rest of the paper. Next, the language XDMap [19] is discussed. The language is used to specify XML schema mappings. We analyze its syntax and semantics. Then, we propose an algorithm for generating automappings. Finally, we show how missing data can be discovered while a mapping is executed. The last subsection concludes the paper.

1.2 Skolem functions, constraints, XML trees and XML schemas

A *Skolem function* returns a uniquely defined values for its arguments. Each of its invocation without arguments generates a new object. If it is invoked more than once for the same arguments it creates a new object only by the first invocation, by consecutive invocations it returns the identifier of the object created by the first evaluation. A concept of using Skolem functions for creation and manipulation object identifiers has been previously proposed in ILOG [11] and in [1, 10]. Recently, Skolem functions have been also used to schema mappings. For example, in Clio [23] they are used for generating missing target values if the target element cannot be null (e.g. components of keys), in [28] are used in a query rewriting based on data mapping.

In XDMap we consider Skolem functions with text-valued arguments, and they are used in two contexts:

1. To compute a string value for the given function name and its arguments. If f is a Skolem function name and $a_1, ..., a_m$ are string values, then the value of the Skolem term $f(a_1, ..., a_m)$ is the string "$f(a_1, ..., a_m)$", called a *term value*, obtained as the concatenation of the function name, its arguments, parenthesis, and separating commas. Further on quotation marks surrounding term values will be omitted.
2. To generate a node (a node identifier) in created resulting XML (data) trees. In this context a Skolem term $F_P(a_1, ..., a_m)$ will be used to express one-to-one relationship between a tuple $(a_1, ..., a_m)$ of key paths values from a source XML tree and the set of nodes labeled with P in a target tree.

We view an XML data as an ordered node-labeled unranked tree (XML tree). We assume that except for simple text values also Skolem term values may be assigned to text (leaf) nodes.

Let *Lab* be a countably infinite set of labels (names), $\mathcal{F}$ be a countably infinite set of Skolem function names, *Str* be a set of string values, *Term* be a set of term values of the form $f(a_1, ..., a_m)$, where $f \in \mathcal{F}$ and $a_i \in Str$, and *Did* be a set of document identifiers (URI addresses or file names). Attributes are modeled as elements.

Definition 1. *Let $L \subset Lab$ be a finite set of labels. An XML tree is a tuple*

$$I = (r, N^e, N^t, child, \leq, \lambda, \nu, \delta), \tag{1.1}$$

where:

1. *r is a distinguished root node, N^e is a finite set of element nodes, and N^t is a finite set of text nodes;*
2. *$child \subseteq \{r\} \cup N^e \times N^e \cup N^t$ – an acyclic binary relation introducing tree structure into (r, N^e, N^t) such that:*
   - *the root has only one child (this child is the outermost element), i.e. $(r, n) \in child \wedge (r, n') \in child \Rightarrow n = n' \wedge n \in N^e$;*
   - *each element node must have a child, i.e: $(n, n') \in child \wedge n' \in N^e \Rightarrow \exists n''(n'' \in (N^e \cup N^t) \wedge (n', n'') \in child)$;*
3. *$\leq$ – a total ordering relation on the set of nodes;*
4. *$\lambda : N^e \to L$ – a function labeling element nodes, the label $l = \lambda(n)$ is the type of n;*
5. *$\nu : N^t \to Str \cup Term$ – a function labeling text nodes with their text values (i.e. string or term values);*
6. *$\delta : \{r\} \to Did$ – a function assigning the document identifier to the root.*

In Fig. 1.1 there are three XML trees I_1, I_2 and I_3. The meaning of labels are: author (A), name (N) and university (U) of the author; paper (P) title (T), year (Y) of publication and the conference (C) where the paper has been presented. Elements labeled with R and K are used to join authors with their papers. Root nodes are not shown explicitly but we assume that they precede the outer most elements and are labeled with I_1, I_2, and I_3, respectively.

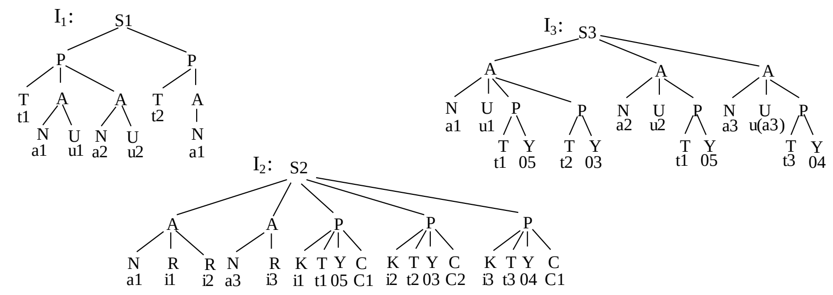

Fig. 1.1. Sample XML trees

XML schema defines both a structure and constraints for XML data. We will consider the following three kinds of constraints: *keys*, *key references*, and *value dependencies.*

1. *Key constraints.* We use a formal approach to keys for XML proposed by Buneman *et al.* [4]. A *key constraint* (or *key*) is an expression of the form

$$\kappa = (P, (P', (P_1, ..., P_k))),$$

where $P/P'/P_i$ is a valid path for every $i = 1, ..., k$. The path P is called the *context path*, P' is called the *target path*, and $P_1, ..., P_k$ are called the *key paths* of κ. When $P = \epsilon$, we call κ an *absolute key* (then ϵ denotes the root), otherwise κ is called a *relative key.* In general, a path P over a set L of labels is an expression with the XPath syntax [26]: $P ::= \epsilon \mid l \mid P/l$, where ϵ is the empty path, $/l$ abbreviates `child::`l and selects the l element children of the context node. An XML tree I satisfies a key κ, denoted by $I \models \kappa$, if any subtree denoted by P' in the context determined by P is uniquely identified by the tuple $(s_1, ..., s_k)$ of text values of the tuple $(P_1, ..., P_k)$ of key paths. For example: $I_1 \models (/S1/P, T)$, $I_1 \models (/S1/P, (A, (N)))$ $I_1 \models (/S1/P/A, (U, (\epsilon)))$, where the last expression indicates that the node of type U is identified only by itself.

2. *A key reference* or *keyref* is an expression of the form

$$\rho = \kappa \ \textbf{ref} \ \kappa',$$

where $\kappa = (P, (P', (P_1, ..., P_k)))$ and $\kappa' = (P, (P'', (P'_1, ..., P'_k)))$ are both key expressions. The key reference κ **ref** κ' defines a *foreign key* κ that refers to a *primary key* κ'. An XML tree I satisfies ρ, denoted $I \models \rho$, if:

- $I \models \kappa'$, i.e. κ' is a key satisfied in I, and
- for each node n in the set determined by P and for any value $(s_1, ..., s_k)$ of the key paths $(P_1, ..., P_k)$ in a node n' in the set reachable from n via P', there is exactly one node n'' in the set reachable from n via P'' in which $(P'_1, ..., P'_k)$ has the value equal to $(s_1, ..., s_k)$.
  For example: $I_2 \models (\epsilon, (/S2/A, (R)))$ **ref** $(\epsilon, (/S2/P, (K)))$.

3. *A value dependency* is an expression of the form

$$\tau = P/l/P' = f(P_1, ..., P_n),$$

that defines functional dependency between a text values denoted by $P/l/P'$ and a tuple of text values determined by the tuple $(P/l/P_1, ..., P/l/P_n)$ of paths. An XML tree I satisfies a a value dependency τ, $I \models \tau$, if:
- $I \models (P, (l, (P_1, ..., P_n))$,
- a text value of the path $P/l/P'$ functionally depends on the tuple of values determined by $(P/l/P_1, ..., P/l/P_n)$.

We use a Skolem function name $f \in \mathcal{F}$ to distinguish two different dependencies having the same set of determining paths. For example: $I_1 \models /S1/P/A/U = u(N)$, $I_2 \models /S2/P/Y = y(T)$, $I_2 \models /S2/P/C = c(T)$, $I_3 \models /S3/A/U = u(N)$, $I_3 \models /S3/A/P/Y = y(T)$.

Definition 2. *An XML schema over $(L, \mathcal{F})$ is a tuple*

$$\mathbf{S} = (top, Seq, Key, Keyref, Valdep), \tag{1.2}$$

where:

- *top is the distinguished label of the outermost element, $top \in L$,*
- *Seq is a function from L to regular expressions over $L - \{top\}$ defined by the grammar $e ::= \epsilon \mid l \mid e|e \mid ee \mid e? \mid e+ \mid e*$;*
- *Key assigns a key constraint $(P, (l, (P_1, ..., P_k)))$ to any label $l \in L - \{top\}$;*
- *$Keyref$ assigns key references, $(P, (P'/l, (P_1, ..., P_k)))$ **ref** $Key(l')$, to some labels in $l \in L - \{top\}$.*
- *$Valdep$ assigns a set of value dependencies, $P/l/P' = f(P_1, ..., P_n)$, to some labels in $l \in L - \{top\}$.* □

In Fig. 1.2 there are three XML data schemas (schema trees) $\mathbf{S}_1$, $\mathbf{S}_2$, and $\mathbf{S}_3$. Instances of these schemas are XML trees I_1, I_2 and I_3 in Fig. 1.1, respectively. These schema trees specify only structural part of an XML schema.

The full description of a schema corresponding to $\mathbf{S}_1$ from Fig. 1.2 written in XML Schema is presented in Fig. 1.3.

Definition 3. *An XML tree $I = (r, N^e, N^t, child, \leq, \lambda, \nu, \delta)$ conforms to the XML schema $\mathbf{S} = (top, Seq, Key, Keyref, Valdep)$, denoted by $I \models \mathbf{S}$, if:*

1. *For the root node r, $\delta(r)$ is defined, and for any text node $n \in N^t$, $\nu(n)$ is defined.*
2. *If $(r, n) \in child$, then $\lambda(n) = top$.*

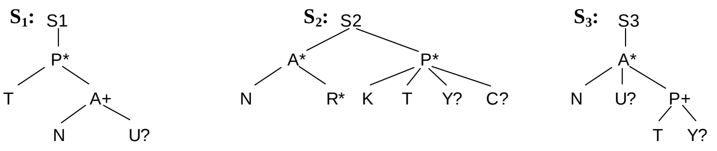

Fig. 1.2. Sample schemas describing structural part of instances from Fig. 1.1

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="S1">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="P"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="P">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="T" type="xs:string"/>
        <xs:element ref="A" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="PKey">
      <xs:selector xpath="."/>
      <xs:field xpath="T"/>
    </xs:key>
  </xs:element>
  <xs:element name="A">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="N" type="xs:string"/>
        <xs:element name="U" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="AKey">
      <xs:selector xpath="."/>
      <xs:field xpath="N"/>
    </xs:key>
    <xs:valdep>
      <xs:dependent xpath="U"/>
      <xs:function name="u"/>
      <xs:argument xpath="N"/>
    </xs:valdep>
  </xs:element>
</xs:schema>
```

Fig. 1.3. Schema of $\mathbf{S}_1$ in XML Schema extended with `<xs:valdep>` declaration

3. *For any node n in I with children $(n_1, ..., n_m)$ such that $n_1 < ... < n_m$, if $\lambda(n) = l$, then the sequence $\lambda(n_1)...\lambda(n_m)$ is a word of the language defined by the regular expression $Seq(l)$.*
4. *For any label $l \in L$*
   - *if a key constraint $Key(l)$ is defined for l, then: $I \models Key(l)$,*
   - *if a key reference $Keyref(l)$ is defined for l, then: $I \models Keyref(l)$,*
   - *if a value dependency $\tau \in Valdep(l)$, then: $I \models \tau$.*

1.3 XML schema mapping - basic ideas

In a general setting of relational data exchange [9, 8, 15], a schema mapping is a triple $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, where $\mathbf{S}$ and $\mathbf{T}$ are, respectively, source and target schemas, and Σ is a set of formulas of some logical formalism over $(\mathbf{S}, \mathbf{T})$.

The formulas in Σ are often specified using *source-to-target tuple-generating dependencies* [2], or *STDs*, that express the relationship between $\mathbf{S}$ and $\mathbf{T}$. They have been used to formalize relational data exchange by Fagin et al [8, 9]. They have also been investigated as GLAV assertions in data integration scenario [12]. A *STD* is a first-order formula of the form [2]

$$\forall \mathbf{x}(\phi_S(\mathbf{x}) \Rightarrow \exists \mathbf{y} \psi_T(\mathbf{x}, \mathbf{y})), \tag{1.3}$$

where $\phi_S(\mathbf{x})$ is a conjunction of atomic formulas over $\mathbf{S}$, and $\psi_T(\mathbf{x}, \mathbf{y}))$ is a conjunction of atomic formulas over $\mathbf{T}$.

In data exchange the following problem is considered [3, 8]: given an instance I over the source schema $\mathbf{S}$, find an instance J over the target schema $\mathbf{T}$ such that the pair $\langle I, J \rangle$ satisfy the *STDs* in Σ. Such an instance J is called a *solution* for I under $\mathcal{M}$.

In general, a mapping specification in XDMap conforms to the general form of *STDs* and has the form [19]:

$$\forall \mathbf{x}(G(\mathbf{x}) \wedge \Phi(\mathbf{x}) \Rightarrow \exists \mathbf{y} C(\mathbf{x}; \mathbf{y}) \wedge \Delta(\mathbf{x}; \mathbf{y})),$$

where $G(\mathbf{x})$ and $\Phi(\mathbf{x})$ are conjunctions of atomic formulas over a source, and $C(\mathbf{x}; \mathbf{y})$ and $\Delta(\mathbf{x}; \mathbf{y})$ are conjunctions of atomic formulas over a target. $G(\mathbf{x})$ defines source variables over source tree, $\Phi(\mathbf{x})$ restricts values of variables, $C(\mathbf{x}; \mathbf{y})$ specifies constraints (value dependencies) over target tree, and $\Delta(\mathbf{x}; \mathbf{y})$ defines child-parent relationships between nodes and, if necessary, also text values for text nodes in the target tree.

Definition 4. *Let a source schema* $\mathbf{S} = (\mathbf{S}_1, ..., \mathbf{S}_n)$ *be a sequence of source XML schemas over* $(L_{\mathbf{S}_i}, \mathcal{F}_{\mathbf{S}_i})$, *respectively, and* $\mathbf{T}$ *be a target XML schema over* $(L_{\mathbf{T}}, \mathcal{F}_{\mathbf{T}})$. *A mapping* $\mathcal{M}_{\mathbf{S},\mathbf{T}}$ *in XDMap from* $\mathbf{S}$ *into* $\mathbf{T}$ *is defined as follows:*

$$\begin{aligned} \mathcal{M}_{\mathbf{S},\mathbf{T}} ::= (G, \Phi, C, \Delta)(\mathbf{x}; \mathbf{y}) := &\ \mathbf{foreach}\ G(\mathbf{x}) \\ &\ \mathbf{where}\ \Phi(\mathbf{x}) \\ &\ \mathbf{when}\ C(\mathbf{x}; \mathbf{y}) \\ &\ \mathbf{exists}\ \Delta(\mathbf{x}; \mathbf{y}) \end{aligned}$$

where

1. $G(\mathbf{x})$ *is a list of variable definitions over source schemas in* $\mathbf{S}$*, a definition of a variable* x *is an expression of the form:* x $\underline{\text{in}}$ P *or* x $\underline{\text{in}}$ x'/P*;*
2. $\Phi(\mathbf{x})$ *is a conjunction of restrictions of the form:* $x = x'$ *over* $\mathbf{x}$*;*
3. $C(\mathbf{x}; \mathbf{y})$ *is a list of target value dependencies of the form:* $x = f(\mathbf{x})$ *or* $y = f(\mathbf{x})$*, where* $x \in \mathbf{x}$*,* $y \in \mathbf{y}$*;*

4. *$\Delta(\mathbf{x};\mathbf{y})$ is a conjunction $\delta_1 \wedge \cdots \wedge \delta_m$ of formulas of the form* $F_{P/l}(\mathbf{x}';\mathbf{y}')$ **in** $F_P(\mathbf{x}'';\mathbf{y}'')/l$[**with** z], *where*
   - *$F_{P/l}(\mathbf{x}';\mathbf{y}')$ is a Skolem term, where P/l is a path in $\mathbf{T}$;*
   - *$F_P(\mathbf{x}'';\mathbf{y}'')/l$ is a target path expression, where l is a label in $\mathbf{T}$;*
   - *$(\mathbf{x}';\mathbf{y}') \subseteq (\mathbf{x};\mathbf{y})$, $(\mathbf{x}'';\mathbf{y}'') \subseteq (\mathbf{x}';\mathbf{y}')$, $z \in (\mathbf{x}';\mathbf{y}')$.*

A mapping from $\mathbf{S}_1$ into $\mathbf{S}_3$ is given in Fig. 1.4.

```
$\mathcal{M}_{13}(x_T, x_N, x_U; y_Y) =$
    foreach $(x_T, x_N, x_U)$ $\underline{\text{in}}$ $\mathbf{S}_1$
    where true
    when $x_U = u(x_N), y_Y = y(x_T)$
    exists
        $F_{/S3}()$ in $F_{()}()/S3$
        $F_{/S3/A}(x_N)$ in $F_{/S3}()/A$
        $F_{/S3/A/N}(x_N)$ in $F_{/S3/A}(x_N)/N$ with $x_N$
        $F_{/S3/A/U}(x_N, x_U)$ in $F_{/S3/A}(x_N)/U$ with $x_U$
        $F_{/S3/A/P}(x_N, x_T)$ in $F_{/S3/A}(x_N)/P$
        $F_{/S3/A/P/T}(x_N, x_T)$ in $F_{/S3/A/P}(x_N, x_T)/T$ with $x_T$
        $F_{/S3/A/P/Y}(x_N, x_T, y_Y)$ in $F_{/S3/A/P}(x_N, x_T)/Y$ with $y_Y$
```

Fig. 1.4. Mapping $\mathcal{M}_{13}$ from $\mathbf{S}_1$ into $\mathbf{S}_3$

Note that the definition of text variables, **foreach** (x_T, x_N, x_U) $\underline{\text{in}}$ $\mathbf{S}_1$, abbreviates in fact the following definition:

foreach x_{S1} $\underline{\text{in}}$ $/S1, x_P$ $\underline{\text{in}}$ $x_{S1}/P, x_T$ $\underline{\text{in}}$ $x_P/T,$
x_A $\underline{\text{in}}$ $x_P/A, x_N$ $\underline{\text{in}}$ $x_A/N, x_U$ $\underline{\text{in}}$ $x_A/U.$

Some variables, for example x_{S1}, x_P, x_A, are auxiliary variables local in the **foreach** clause, while another, for example x_T, x_N, x_U, are source text variables global in the mapping. The variable y_Y is a target variable defined in the **when** clause as a function of x_T. The variable y_Y is defined only in the **when** clause because instances of $\mathbf{S}_1$ do not provide data about year of publication. However, we know that the year of publication of a paper depends on the title of the paper. This is denoted by the value dependency constraint: $y_Y = y(x_T)$.

Assuming that a schema denotes a set of all its instances, a mapping is an n-ary function that maps a tuple of source instances to a target instance (we will omit subscripts of $\mathcal{M}$ if they are clear from the context):

$$\mathcal{M} : \mathbf{S}_1 \times ... \times \mathbf{S}_m \to \mathbf{T}, \tag{1.4}$$

where for any tuple $(I_1, ..., I_n)$ such that $I_i \models \mathbf{S}_i$, $\mathcal{M}(I_1, ..., I_n) = J \models \mathbf{T}$.

We will pay a special attention to *automappings* which are identity mappings from a schema onto itself, i.e. a mapping $\mathcal{M}$ is the automapping over $\mathbf{S}$ iff $\mathcal{M} : \mathbf{S} \to \mathbf{S}$, where for each $I \in \mathbf{S}$, $\mathcal{M}(I) = I$.

1.4 Semantics of mapping rules

We propose a semantics for the XDMap in the operational way. We state how, for a given set $(\Omega, \leq)$ of bindings of variables in $(\mathbf{x};\mathbf{y})$ into text values of an XML tree, determined by the **foreach/where/when** clauses, a set of mapping rules is executed and how the resulting target instance tree is created. The result tree must be ordered, so the encoding of the tree's nodes is a challenging issue. To encode ordering of the XML tree's node a variaty of order encoding methods is possible [13, 16]. We will use the Dewey order encoding, where each node n is assigned a vector $Pos(n)$ that represents the path from the document's root to n. For example, if $Pos(n) = 0.1.3.2$ then n is the second child of a node n', where n' is the third child of the outermost element that is the first element of the document (0 represents the root). The Dewey vector provides information about both the relative position of a node within children of the node's parent, and the absolute position of the node within the document (document's XML tree).

We will construct the Dewey vector using the fact that bindings $\omega \in \Omega$ are totally ordered. Then we assume the following position function:

- $Pos(r) = 0$, if r is the root,
- $Pos(n) = Pos(n').\omega$, where n' is the parent of n and ω is a binding for variables $\mathbf{x};\mathbf{y}$ in an expression $F_P(\mathbf{x};\mathbf{y})$ used to generate the node n. If the set of variables is empty (that is the case for the outermost element) we assume that ω equals 1.

Semantics for XDMap is defined as follows:

Definition 5. *Let $\mathcal{M} = (G, \Phi, C, \Delta)(\mathbf{x};\mathbf{y})$ be a mapping over a schema $\mathbf{S}$ and I be an instance of $\mathbf{S}$. A target instance $J = \mathcal{M}(I)$,*

$$J = (r, N^e, N^t, child, \leq, \lambda, \nu, \delta), \tag{1.5}$$

is obtained by means of the semantic functions $\mathbb{E}, \mathbb{M}, \mathbb{R}$, and $\mathbb{N}$ in the following way:

1. $r = \mathbb{N}(F_{@doc}())$, *and* $\delta(r) = @doc$, $Pos(r) = 0$.
2. $n = \mathbb{N}(F_{P/l}(\mathbf{x};\mathbf{y}))(\omega) = F_{P/l}(\omega(\mathbf{x});\omega(\mathbf{y}))$, $n \in N^e$, $\lambda(n) = l$.
3. $\mathbb{R}(F_{P/l}(\mathbf{x};\mathbf{y})$ **in** $F_P(\mathbf{x}';\mathbf{y}')/l$ [**with** x'']$)(\omega)=$
   $\mathbb{R}(\mathbb{N}(F_{P/l}(\omega(\mathbf{x});\omega(\mathbf{y}))$ **in** $F_P(\omega(\mathbf{x}');\omega(\mathbf{y}'))/l$ [**with** $\omega((x'')$])
   if $n = \mathbb{N}(F_{P/l}(\mathbf{x};\mathbf{y})(\omega)$, $n' = \mathbb{N}(F_P(\mathbf{x}';\mathbf{y}')(\omega)$, $v = \omega(x'')$,
   then $(n', n) \in child$, $Pos(n) = Pos(n').\omega$,
   $n'' = newId() \in N^t$, $(n, n'') \in child$, $\nu(n'') = v$, $Pos(n'') = Pos(n).1$
4. $n \leq n' \Leftrightarrow Pos(n) \leq Pos(n')$.
5. $\mathbb{M}(\Delta(\mathbf{x};\mathbf{y}))(\Omega) = \{\mathbb{R}(\delta(\mathbf{x};\mathbf{y}))(\omega) \mid \delta \in \Delta,\ \omega \in \Omega\}$.
6. $\mathbb{E}(\mathcal{M}(\mathbf{x};\mathbf{y}))(I) = \mathbb{M}(\Delta(\mathbf{x};\mathbf{y}))(((G, \Phi, C)(\mathbf{x};\mathbf{y}))(I)) = \mathbb{M}(\Delta(\mathbf{x};\mathbf{y}))(\Omega)$, *where Ω is a totally ordered set of bindings of variables $(\mathbf{x};\mathbf{y})$ determined by (G, Φ, C).* $\square$

According to the definition of semantics, a new XML tree is created as follows:

1. The execution of $F_{@doc}()$ produces a new root node r. The node is associated with a provided name $@doc$ that is a unique name (URI or file name) of the newly created document. The root node r gets the ordering number 0 and precedes the first node of the created tree.
2. Execution of $F_{P/l}(\omega(\mathbf{x});\omega(\mathbf{y}))$ produces a new element node of type l.
3. The expression $\mathbb{R}(n, n'/l, [\ v\ ])$ acts as follows: n is assumed to be a child of type l of n', the order position of n is set to $Pos(n').\omega$; if the third optional argument v is given, then a new text node n'' is created, n'' becomes a child of n, and v is assigned to n'' as its text value, the order position of n'' is set to $Pos(n).1$;
4. The total ordering on nodes coincides with the total ordering on Dewey vectors.
5. The execution from p. (3) is carried out for any mapping rule $\delta \in \Delta$ and for any binding $\omega \in \Omega$.
6. Execution of $\mathcal{M}(I)$ proceeds in two phases. First, a set Ω of bindings is determined, and then Ω is used in execution of the set Δ of mapping rules by the semantic function $\mathbb{M}$.

1.5 Using constraints to generate automappings

Information provided by an XML schema $\mathbf{S}$ my be used to generate the automapping that transform instances of $\mathbf{S}$ onto themselves, Algorithm 1.

Algorithm 1 *Automapping generation from a schema*
Input : An XML schema $\mathbf{S} = (top, Seq, Key, Keyref, Dep)$ over $(L, \mathcal{F})$.
Output : Automapping
$\mathcal{M} = (\mathbf{foreach}\ G, \mathbf{where}\ \Phi, \mathbf{when}\ C, \mathbf{exists}\ \Delta)$ over $\mathbf{S}$.

```
1. $G := \{x_{top}\ \underline{\text{in}}\ /top\}$; $\Phi := \{\mathbf{true}\}$;
   $C := \emptyset$; $\Delta := \{F_{top}()\ \mathbf{in}\ F_{(@doc)}()/top\}$;
   $\mathbf{x}_{top} := (x_{/top})$
2. foreach $l \in L$ begin
   case $Key(l)$ of
      $(P, (l, (P_1, ..., P_m))),\ m > 0$ :
         $G := G \cup \{x_{P/l}\ \underline{\text{in}}\ x_P/l, ..., x_{P/l/P_m}\ \underline{\text{in}}\ x_{P/l}/P_m\}$
         $\mathbf{x}_{P/l} := \mathbf{x}_P \circ (x_{P/l/P_1}, ..., x_{P/l/P_m})$
         $\Delta := \Delta \cup \{F_{P/l}(\mathbf{x}_{P/l})\ \mathbf{in}\ F_P(\mathbf{x}_P)/l\}$
      $(P, (l, \epsilon))$ :
         $G := G \cup \{x_{P/l}\ \underline{\text{in}}\ x_P/l)$
         $\mathbf{x}_{P/l} := \mathbf{x}_P \circ (x_{P/l})$
         $\Delta := \Delta \cup \{F_{P/l}(\mathbf{x}_{P/l})\ \mathbf{in}\ F_P(\mathbf{x}_P)/l\ \mathbf{with}\ x_{P/l}\}$
```

```
endcase
if $Keyref(l) = (P, (P'/l, (P_1, ..., P_k)))$ ref $(P, (P''/l', (P'_1, ..., P'_k)))$ then
    $\Phi := \Phi \cup \{x_{P/P'/l/P_1} = x_{P/P''/l'/P'_1}, ..., x_{P/P'/l/P_m} = x_{P/P''/l'/P'_m}\}$
foreach $P/l/P' = f(P_1, ..., P_m) \in Valdep(l)$
    $C := C \cup \{x_{P/l/P'} = f(x_{P/l/P_1}, ..., x_{P/l/P_m})\}$
end
```

For example, Algorithm 1 generates the automapping $\mathcal{M}_{33}$ for the schema $\mathbf{S}_3$ (Fig. 1.5).

```
$\mathcal{M}_{33}(\mathbf{y})$ = foreach $(y_N, y_U, y_T, y_Y)$ $\underline{\text{in}}$ $\mathbf{S}_3$
          where true
          when $y_U = u(y_N), y_Y = y(y_T)$
          exists
             $F_{/S3}()$ in $F_{()}()/S3$
             $F_{/S3/A}(y_N)$ in $F_{/S3}()/A$
             $F_{/S3/A/N}(y_N)$ in $F_{/S3/A}(y_N)/N$ with $y_N$
             $F_{/S3/A/U}(y_N, y_U)$ in $F_{/S3/A}(y_N)/U$ with $y_U$
             $F_{/S3/A/P}(y_N, y_T)$ in $F_{/S3/A}(y_N)/P$
             $F_{/S3/A/P/T}(y_N, y_T)$ in $F_{/S3/A/P}(y_N, y_T)/T$ with $y_T$
             $F_{/S3/A/P/Y}(y_N, y_T, y_Y)$ in $F_{/S3/A/P}(y_N, y_T)/Y$ with $y_Y$
```

Fig. 1.5. Automapping $\mathcal{M}_{33}$ over $\mathbf{S}_3$

In (a fragment of) the definition of $\mathbf{S}_2$ (Fig. 1.6), the schema specifies the *key* and *keyref* relationships between the K child element of the P element (the *primary key*) and the R child element of the A element (the *foreign key*). For this schema, Algorithm 1 generates the automapping $\mathcal{M}_{22}$ given in Fig. 1.7.

Mappings can be combined by means of some operators giving a result that in turn is a mapping. We have defined three operations: *Match*, *Compose*, and *Merge* in [19]. Some of these operations require specification of a *correspondence* between paths of schemas under consideration. Establishing the correspondence is a crucial task in definition of data mappings [24].

1.6 Using value constraints to infer missing data by executing mappings

In Fig. 1.8 there is an executable mapping, that integrates, by means of the *Merge* operator, instances of schemas $\mathbf{S}_1$ and $\mathbf{S}_2$ under the schema $\mathbf{S}_3$. Now, we focus on the problem of discovering missing values in the process of mapping execution. The discovery is achieved using some inference rules over (partial) bindings of variables.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="S2">
  ...
  </xs:element>
  <xs:element name="A">
   ...
    <xs:keyref name="RKeyref" refer="KKey">
      <xs:selector xpath="."/>
      <xs:field xpath="R"/>
    </xs:keyref>
  </xs:element>
  <xs:element name="P">
    ...
    <xs:key name="KKey">
      <xs:selector xpath="."/>
      <xs:field xpath="K"/>
    </xs:key>
  </xs:element>
</xs:schema>
```

Fig. 1.6. Schema of $\mathbf{S}_2$ expressed in XML Schema language

$$
\begin{aligned}
\mathcal{M}_{22} = \ & \textbf{foreach } (z_N, z_R, z_K, z_T, z_Y, z_C \ \underline{\text{in}} \ \mathbf{S}_2 \\
& \textbf{where } z_R = z_K \\
& \textbf{when } z_K = k(z_N, z_T), z_Y = y(z_T), z_C = c(z_T) \\
& \textbf{exists} \\
& F_{/S2}() \textbf{ in } F_{()}()/S2 \\
& F_{/S2/A}(z_N) \textbf{ in } F_{/S2}()/A \\
& F_{/S2/A/N}(z_N) \textbf{ in } F_{/S2/A}(z_N)/N \textbf{ with } z_N \\
& F_{/S2/A/R}(z_N, z_K) \textbf{ in } F_{/S2/A}(z_N)/R \textbf{ with } z_K \\
& F_{/S2/P}(z_K) \textbf{ in } F_{/S2}()/P \\
& F_{/S2/P/K}(z_K) \textbf{ in } F_{/S2/P}(z_K)/K \textbf{ with } z_K \\
& F_{/S2/P/T}(z_K, z_T) \textbf{ in } F_{/S2/P}(z_K)/T \textbf{ with } z_T \\
& F_{/S2/P/Y}(z_K, z_Y) \textbf{ in } F_{/S2/P}(z_K)/Y \textbf{ with } z_Y \\
& F_{/S2/P/C}(z_K, z_C) \textbf{ in } F_{/S2/P}(z_K)/C \textbf{ with } z_C
\end{aligned}
$$

Fig. 1.7. Automapping $\mathcal{M}_{22}$ over $\mathbf{S}_2$

Execution of $Merge_{\mathbf{S}_3}(\mathbf{S}_1, \mathbf{S}_2)(I_1, I_2)$ consists of the following four steps:

1. *Determining a set Ω of bindings and a set Ω'_Ω of dependent bindings.* Variable specifications in the **foreach** clause over schemas $\mathbf{S}_1$ and $\mathbf{S}_2$ are computed against instances I_1 and I_2, respectively, and produce two sets Ω_1 and Ω_2 of partially defined bindings. By Ω we denote the union of Ω_1 and Ω_2. By Ω'_Ω we denote a set of dependent bindings for dependent variables (specified in the **when** clause). A binding $\omega'_\omega \in \Omega'_\Omega$ binds a term value to a dependent variable, e.g. $\omega'_\omega(x_U) = u(\omega(x_N))$. The set Ω of all bindings for all variables, and the set Ω'_Ω of dependent bindings for all

```
$Merge_{\mathbf{S}_3}(\mathbf{S}_1, \mathbf{S}_2) =$
    foreach $(x_T, x_N, x_U)$ in $\mathbf{S}_1, (z_N, z_R, z_K, z_T, z_Y)$ in $\mathbf{S}_2$
    where  $z_R = z_K$
    when   $x_U = u(x_N), y_Y = y(x_T), v_U = u(z_N), z_Y = y(z_T)$
    exists
        (1) $F_{/S3}()$ in $F_{()}()/S3$
        (2) $F_{/S3/A}(x_N)$ in $F_{/S3}()/A$
            $F_{/S3/A}(z_N)$ in $F_{/S3}()/A$
        (3) $F_{/S3/A/N}(x_N)$ in $F_{/S3/A}(x_N)/N$ with $x_N$
            $F_{/S3/A/N}(z_N)$ in $F_{/S3/A}(z_N)/N$ with $z_N$
        (4) $F_{/S3/A/U}(x_N, x_U)$ in $F_{/S3/A}(x_N)/U$ with $x_U$
            $F_{/S3/A/U}(z_N, v_U)$ in $F_{/S3/A}(z_N)/U$ with $v_U$
        (5) $F_{/S3/A/P}(x_N, x_T)$ in $F_{/S3/A}(x_N)/P$
            $F_{/S3/A/P}(z_N, z_T)$ in $F_{/S3/A}(z_N)/P$
        (6) $F_{/S3/A/P/T}(x_N, x_T)$ in $F_{/S3/A/P}(x_N, x_T)/T$ with $x_T$
            $F_{/S3/A/P/T}(z_N, z_T)$ in $F_{/S3/A/P}(z_N, z_T)/T$ with $z_T$
        (7) $F_{/S3/A/P/Y}(x_N, x_T, y_Y)$ in $F_{/S3/A/P}(x_N, x_T)/Y$ with $y_Y$
            $F_{/S3/A/P/Y}(z_N, z_T, z_Y)$ in $F_{/S3/A/P}(z_N, z_T)/Y$ with $z_Y$
```

Fig. 1.8. A mapping specifying merging of $\mathbf{S}_1$ and $\mathbf{S}_2$ under $\mathbf{S}_3$

dependent variables, are shown in Fig. 1.9(1). Bindings in Ω are partial functions because some bindings for some variables may be undefined – we denote this by $\perp$. For example, $\omega_3(x_U) = \perp$.

2. *Expanding bindings from Ω.*
   If $\omega \in \Omega$ and $\omega(x) = \perp$, i.e. ω is not defined for x, then we assume
   $$\omega(x) := \omega'_\omega(x). \tag{1.6}$$
   In this way we assign a term value to a variable for which there is no explicit binding. For example, $\omega_3(x_U) := \omega'_{\omega_3}(x_U) = u(\omega_3(x_N)) = u(a1)$. In Fig. 1.9(2) there is the result of expanding bindings from Ω.
3. *Resolving term values in bindings.*
   In this step we try to discover text values for these variables to which term values have been assigned. We say that such variables have *missing values*. To achieve this the following inference rule is applied:
   $$\omega'_{\omega_1}(x_1) = \omega'_{\omega_2}(x_2) \Rightarrow \omega_1(x_1) := \omega_2(x_2) \tag{1.7}$$
   For example, in this way we can obtain that $\omega_1(y_Y) = 05$. This is obtained as follows: $\omega'_{\omega_1}(y_Y) = \omega'_{\omega_4}(z_Y)$, so using the rule (1.7) we have $\omega_1(y_Y) := \omega_4(z_Y) = 05$. Note that $\omega_6(v_U)$ can not be resolved. The resolved set of bindings is shown in Fig. 1.9(3).

After preparing an expanded and resolved set Ω of bindings, the **exist** clause of the mapping can be executed. For the mapping in Fig. 1.8 the execution proceeds as follows (we discuss only some representative mapping expressions):

(1) Bindings $\Omega = \Omega_1 \cup \Omega_2$:

Ω	x_T	x_N	x_U	y_Y	z_N	z_R	z_K	z_T	z_Y	v_U
ω_1	$t1$	$a1$	$u1$	$\perp$						
ω_2	$t1$	$a2$	$u2$	$\perp$						
ω_3	$t2$	$a1$	$\perp$	$\perp$						
ω_4					$a1$	$i1$	$i1$	$t1$	05	$\perp$
ω_5					$a1$	$i2$	$i2$	$t2$	03	$\perp$
ω_6					$a3$	$i3$	$i3$	$t3$	04	$\perp$

Dependent bindings Ω'_Ω:

Ω'_Ω	x_U	y_Y	v_U	z_Y
ω'_{ω_1}	$u(a1)$	$y(t1)$		
ω'_{ω_2}	$u(a2)$	$y(t1)$		
ω'_{ω_3}	$u(a1)$	$y(t2)$		
ω'_{ω_4}			$u(a1)$	$y(t1)$
ω'_{ω_5}			$u(a1)$	$y(t2)$
ω'_{ω_6}			$u(a3)$	$y(t3)$

(2) Set Ω of bindings after expanding:

Ω	x_T	x_N	x_U	y_Y	z_N	z_R	z_K	z_T	z_Y	v_U
ω_1	$t1$	$a1$	$u1$	$y(t1)$						
ω_2	$t1$	$a2$	$u2$	$y(t1)$						
ω_3	$t2$	$a1$	$u(a1)$	$y(t2)$						
ω_4					$a1$	$i1$	$i1$	$t1$	05	$u(a1)$
ω_5					$a1$	$i2$	$i2$	$t2$	03	$u(a1)$
ω_6					$a3$	$i3$	$i3$	$t3$	04	$u(a3)$

(3) Set Ω of bindings after expanding and resolving:

Ω	x_T	x_N	x_U	y_Y	z_N	z_K	z_T	z_Y	v_U
ω_1	$t1$	$a1$	$u1$	05					
ω_2	$t1$	$a2$	$u2$	05					
ω_3	$t2$	$a1$	$u1$	03					
ω_4					$a1$	$i1$	$t1$	05	$u1$
ω_5					$a1$	$i2$	$t2$	03	$u1$
ω_6					$a3$	$i3$	$t3$	04	$u(a3)$

Fig. 1.9. Determining, expanding and resolving a set Ω of bindings during execution of the mapping $Merge_{\mathbf{S}_3}(\mathbf{S}_1, \mathbf{S}_2)$ on the pair of instances (I_1, I_2)

(1) Two new nodes are created, the root r and the node n of the outermost element of type $/S3$, as results of Skolem functions $F_{()}()$ and $F_{/S3}()$, respectively. The node n is a child of type $S3$ of r.
(2) A new node n' for any distinct value of x_N is created. Each such node has the type $/S3/A$ and is a child of type A of the node n created by $F_{/S3}()$.
(3) For any distinct value of x_N a new node n'' of type $/S3/A/N$ is created. Each such node is a child of type N of the node created by invocation of $F_{/S3/A}(x_N)$ in (2) for the same value of x_N. Because n'' is a leaf, it obtains the text value equal to the current value of x_N.
(4) Analogously for the rest of the specification.

As the result, we obtain the instance I_3 depicted in Fig. 1.1.

1.7 Conclusion

In the paper the problem of schema mapping is considered, which occurs in many data management systems such as XML data exchange, XML data integration or e-commerce applications. Our solution to this problem relies on the automatic generation of semantics-preserving schema mappings. We discussed how automappings could be generated using schema constraints, such as keys, key references and value dependencies, defined in XML Schema. The mapping specification language XDMap is discussed. Mapping rules in XDMap are defined in conformity with source-to-target data generating dependencies. Skolem functions are used in these rules to express both functional dependencies between some text values in the target instance, and between tuples of key path values in the sources and subtrees in the target. Mappings between two schemas can be generated automatically from their automappings and correspondences between schemas. Automappings represent schemas, so operations over schemas and mappings can be defined and performed in a uniform way. We show how constraints on values can be used to infer some missing data. Our techniques can be applied in various XML data exchange scenarios, and are especially useful when the set of data sources change dynamically (e.g. in P2P environment) [25, 5] or when merging data from heterogeneous sources is needed [22]. The method proposed in the paper is under implementation in a system for semantic integration of XML data in P2P environment using schemas and ontologies [6, 19].

References

1. Abiteboul, S., Buneman, P., Suciu, D.: *Data on the Web. From Relational to Semistructured Data and XML*, Morgan Kaufmann, San Francisco, 2000.
2. Abiteboul, S., Hull, R., Vianu, V.: *Foundations of Databases*, Addison-Wesley, Reading, Massachusetts, 1995.
3. Arenas, M., Libkin, L.: XML Data Exchange: Consistency and Query Answering, *PODS Conference*, 2005, 13–24.
4. Buneman, P., Davidson, S. B., Fan, W., Hara, C. S., Tan, W. C.: Reasoning about keys for XML, *Information Systems*, **28**(8), 2003, 1037–1063.
5. Calvanese, D., Giacomo, G. D., Lenzerini, M., Rosati, R.: Logical Foundations of Peer-To-Peer Data Integration., *Proc. of the 23rd ACM SIGMOD Symposium on Principles of Database Systems (PODS 2004)*, 2004, 241–251.
6. Cybulka, J., Meissner, A., Pankowski, T.: Schema- and Ontology-Based XML Data Exchange in Semantic E-Business Applications, *Business Information Systems, BIS 2006, Lecture Notes in Informatics, Vol.85*, 2006, 429–441.
7. Fagin, R., Kolaitis, P. G., Miller, R. J., Popa, L.: Data Exchange: Semantics and Query Answering., *ICDT 2003*, Lecture Notes in Computer Science 2572, Springer, 2002, 207–224.
8. Fagin, R., Kolaitis, P. G., Popa, L.: Data exchange: getting to the core., *ACM Trans. Database Syst.*, **30**(1), 2005, 174–210.

9. Fagin, R., Kolaitis, P. G., Popa, L., Tan, W. C.: Composing Schema Mappings: Second-Order Dependencies to the Rescue., *PODS*, 2004, 83–94.
10. Fernandez, M. F., Florescu, D., Kang, J., Levy, A. Y., Suciu, D.: Catching the Boat with Strudel: Experiences with a Web-Site Management System., *SIGMOD Conference*, 1998, 414–425.
11. Hull, R., Yoshikawa, M.: ILOG: Declarative Creation and Manipulation of Object Identifiers., *VLDB*, 1990, 455–468.
12. Lenzerini, M.: Data Integration: A Theoretical Perspective., *PODS*, 2002, 233–246.
13. Li, Q., Moon, B.: Indexing and Querying XML Data for Regular Path Expressions, *Proc. of the 27th International Conference on Very Large Data Bases, VLDB 2001, Rome, Italy*, 2001, 361–370.
14. Melnik, S., Bernstein, P. A., Halevy, A. Y., Rahm, E.: Supporting Executable Mappings in Model Management., *SIGMOD Conference*, 2005, 167–178.
15. Nash, A., Bernstein, P. A., Melnik, S.: Composition of Mappings Given by Embedded Dependencies., *PODS*, 2005, 172–183.
16. O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: Insert-Friendly XML Node Label, *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004, 903–908.
17. Pankowski, T.: A High-Level Language for Specifying XML Data Transformations, *Advances in Databases and Information Systems ADBIS 2004*, Lecture Notes in Computer Science **3255**, Springer, 2004, 159–172.
18. Pankowski, T.: Specifying Schema Mappings for Query Reformulation in Data Integration Systems, *Atlantic Web Intelligence Conference - AWIC'2005*, Lecture Notes in Computer Science **3528**, Springer, 2005, 361–365.
19. Pankowski, T.: Management of executable schema mappings for XML data exchange, *Database Technologies for Handling XML Information on the Web, EDBT 2006 Workshops*, Lecture Notes in Computer Science **4254**, Springer, 2006, 264–277.
20. Pankowski, T.: Reasoning About Data in XML Data Integration, *Information Processing and Management of Uncertainty in Knowledge-based Systems, IPMU 2006, Vol 3*, Editions EDK, Paris, 2006, 2506–2513.
21. Pankowski, T.: Integration of XML Data in Peer-To-Peer E-commerce Applications, 5^{th} *IFIP Conference I3E'2005*, Springer, New York, 2005, 481–496.
22. Pankowski, T., Hunt, E.: Data Merging in Life Science Data Integration Systems, *Intelligent Information Systems, New Trends in Intelligent Information Processing and Web Mining*, Advances in Soft Computing, Springer Verlag, 2005, 279–288.
23. Popa, L., Velegrakis, Y., Miller, R. J., Hernández, M. A., Fagin, R.: Translating Web Data., *VLDB*, 2002, 598–609.
24. Rahm, E., Bernstein, P. A.: A survey of approaches to automatic schema matching, *The VLDB Journal*, **10**(4), 2001, 334–350.
25. Tatarinov, I., Halevy, A. Y.: Efficient Query Reformulation in Peer-Data Management Systems., *SIGMOD Conference*, 2004, 539–550.
26. XML Path Language (XPath) 2.0, W3C Working Draft: 2002. www.w3.org/TR/xpath20
27. XML Schema Part 1: Structures: 2004. www.w3.org/TR/xmlschema-1
28. Yu, C., Popa, L.: Constraint-Based XML Query Rewriting For Data Integration., *SIGMOD Conference*, 2004, 371–382.