Management of executable schema mappings for XML data exchange

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
`tadeusz.pankowski@put.poznan.pl`

Abstract. Executable schema mappings between XML schemas are essential to support numerous data management tasks such as data exchange, data integration and schema evolution. The novelty of this paper consists in a method for automatic generation of automappings (automorphisms) from key constraints and value dependencies over XML schemas, and designing algebraic operations on mappings and schemas represented by automappings. During execution of mappings some missing or incomplete data may be inferred. A well-defined executable semantics for mappings and operations on mappings are proposed. A mapping language XDMap to specify XML schema mappings is discussed. The language allows to specify executable mappings that can be used to compute target instances from source instances preserving key constraints and value dependencies. The significance of mappings and operators over mappings is discussed on a scenario of data exchange in a P2P setting.

1 Introduction

Schema mapping is a basic problem for many applications such as data exchange, data integration, P2P databases or e-commerce, where data may be available at many different peers in many different schemas [3, 6, 10, 11, 17, 20]. A schema mapping specifies a constraint that holds between schemas and can be thought of as a relation on instances. Executable mappings are mappings that are able to compute target instances from source instances preserving a set of given constraints [11]. The main contributions of this work are as follows:

1. We propose a method for generating *automappings* over schemas from key and value dependency constraints defined in schemas. Automappings are then used to create mappings between schemas (*Match* operator). Mappings can be combined (using $Compose$ and $Merge$ operators) to give new mappings. We propose a mapping language, called XDMap, to specify mappings.
2. In the process of data transformation some missing or incomplete data, which are not given explicitly in sources, can be deduced based on value dependency constraints enforced by the target schema. This is achieved by representing

missing data by terms defining the constraints. In some cases such terms may be resolved and replaced by the actual data.

The following section discuss the contribution of the paper against related work. The next section shows a scenario of data exchange. Section 4 illustrates the problem of using constraints to mapping specifications and to inferring some missing data. Section 5 describes basic ideas of our approach and proposes syntax and semantics for the mapping language XDMap. Operations on mappings are discussed in Section 6. Section 7 concludes the paper.

2 Related work

We discuss our contribution from the following three points of view.

1. A language for mapping specification. 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, 6, 11, 13]. 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, 19]. In our mapping language XDMap we use Skolem functions with text-valued arguments from a source instance to create nodes (node identifiers) in a target instance. A concept of using Skolem functions for creation and manipulation object identifiers has been previously proposed in ILOG [8] and in [1, 7]. Recently, Skolem functions are also used in some approaches to schema mappings, in Clio [16] are used for generating missing target values if the target element cannot be null (e.g. components of keys), in [19] are used in a query rewriting based on data mapping. Our mapping language can be compared with the mapping language proposed in [19]. However, XDMap is more powerful because we can use arbitrary Skolem functions for intermediate (non leaf-level) nodes, while in [19] these Skolem functions are system generated in a controlled way. Consequently, in our mappings, node generation is controlled by the mapping itself.

2. Generating mappings from key constraints. To define mappings we assume that key and some value constraints are specified within schema (using XML Schema [18] notation). We show how an automapping (a mapping from a schema onto itself) 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 [19]) constraints must be explicitly encoded in the mapping language. This can make difficulties for future management when schemas evolve. To define correspondences between schema elements we use the method proposed in [9, 16] where a correspondence is defined between single elements from a source and a target schema. Establishing of a correspondence may be supported by automated techniques [17].

3. Algebra of mappings. Since a schema is represented by its automapping, we can operate over schemas and mappings in a uniform way. We discuss three operators over mappings (schemas): *Match* – creates a mapping between two schemas (it is a special case of composition), *Compose* – combines two successive mappings, and *Merge* – produces a mapping that merges two source schemas. Operations on mappings, mainly composition, was recently studied in [6, 10–13].

3 A scenario of data exchange

We illustrate XML data exchange on a scenario of a P2P data exchanging system. Suppose there are three peers with schemas $\mathbf{S}_1$, $\mathbf{S}_2$, and $\mathbf{S}_3$, respectively (Fig. 1). Only $\mathbf{S}_2$ and $\mathbf{S}_3$ are associated with data, while $\mathbf{S}_1$ is a mediated (or target) schema that does not store any data. 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. I_2 and I_3 are instances of $\mathbf{S}_2$ and $\mathbf{S}_3$, respectively. In such scenario we meet the problem of data exchange, i.e. computing target instances from source instances [3, 5, 11, 16, 19]. Mappings are needed to perform these functions effectively.

An instance of $\mathbf{S}_1$ can be obtained in different ways (Fig. 2): (1) and (2) by simple transformation of instance I_2 or I_3 by means of mappings $\mathcal{M}_{21}$ or $\mathcal{M}_{31}$; (3) as a merge $\mathcal{M}_{21} \cup \mathcal{M}_{31}$ over instances; (4) by means of composition $(\mathcal{M}_{32} \circ \mathcal{M}_{21})(\mathrm{I}_3)$ (when the $\mathbf{S}_2$'s peer is unavailable); (5) using combination of merge and composition, $((\mathcal{M}_{22} \cup \mathcal{M}_{32}) \circ \mathcal{M}_{21})(\mathrm{I}_2, \mathrm{I}_3)$. This shows that we often need to create new mappings from existing ones [6, 10, 11].

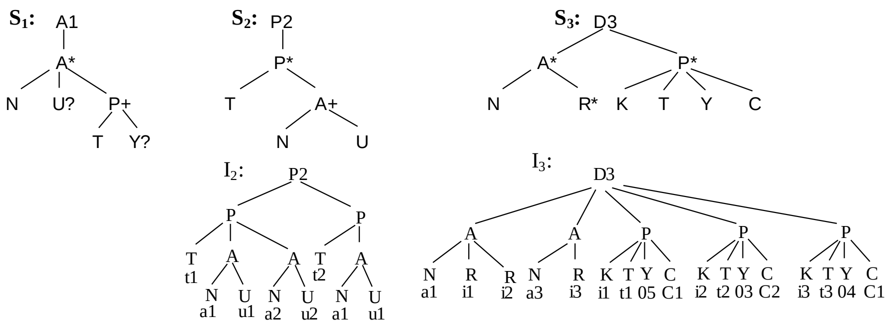
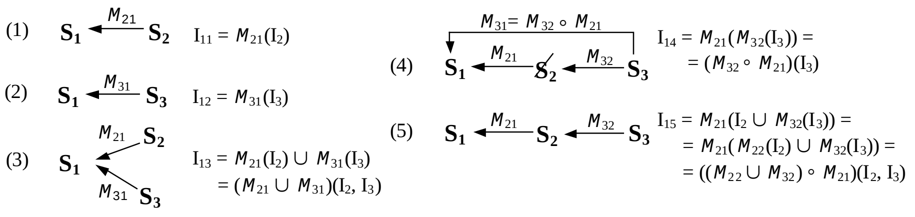

S₁: A1
A*
N U? P+
T Y?
S₂: P2
P*
T A+
N U
S₃: D3
A* P*
N R* K T Y C
I₂: P2
P P
T t1 A A T t2 A
N a1 U u1 N a2 U u2 N a1 U u1
I₃: D3
A A P P P
N a1 R i1 R i2 N a3 R i3 K i1 T t1 Y 05 C C1 K i2 T t2 Y 03 C C2 K i3 T t3 Y 04 C C1

Fig. 1. Schemas: $\mathbf{S}_1, \mathbf{S}_2, \mathbf{S}_3$, and schema instances I_2 and I_3

4 Using constraints for mapping specification

In our approach, we use two kinds of constraints to define mappings, namely:



(1) S1 M21 S2 I11 = M21(I2)
(2) S1 M31 S3 I12 = M31(I3)
(3) S1 M21 S2 M31 S3 I13 = M21(I2) ∪ M31(I3) = (M21 ∪ M31)(I2, I3)
(4) M31= M32 ∘ M21 S1 M21 S2 M32 S3 I14 = M21(M32(I3)) = = (M32 ∘ M21)(I3)
(5) S1 M21 S2 M32 S3 I15 = M21(I2 ∪ M32(I3)) = = M21(M22(I2) ∪ M32(I3)) = = ((M22 ∪ M32) ∘ M21)(I2, I3)

Fig. 2. Scenarios of data exchange – an instance of $\mathbf{S}_1$ may be computed in many ways

1. *Value dependency constraints* (on the target) imposing that a value of a path depends on a tuple of values of other paths. We will declare them in the `<xs:valdep>` section of XML Schema (Fig. 4) that is a non-standard element within XML Schema.
2. *Key constraints* (on a source) stating that a subtree is uniquely identified by a tuple of values of key paths [4, 18]. They are specified within `<xs:key>` and `<xs:keyref>` sections of XML Schema (Fig. 4).

Value dependencies can be used to infer missing data. Suppose we want to transform the instance I_2 under the target schema $\mathbf{S}_1$, i.e. an instance $\mathrm{I}_{11} = \mathcal{M}_{21}(\mathrm{I}_2)$ must be produced (Fig. 3(a)). The original instance provides no data about publication year. However, we know that the publication year (Y) uniquely depends on the title (T) of the paper that is denoted by the constraint $Y = y(T)$, where y is the name of a function mapping titles into publication years. So the term $y(t)$, where t is the title, is assigned to Y as its text value. This forces some elements of type Y to have the same values (Fig. 3(a)). Such constraints are defined within the `<xs:valdep>` section in XML Schema (Fig. 4). Next, a term like $y(t)$ may be resolved using other mappings.

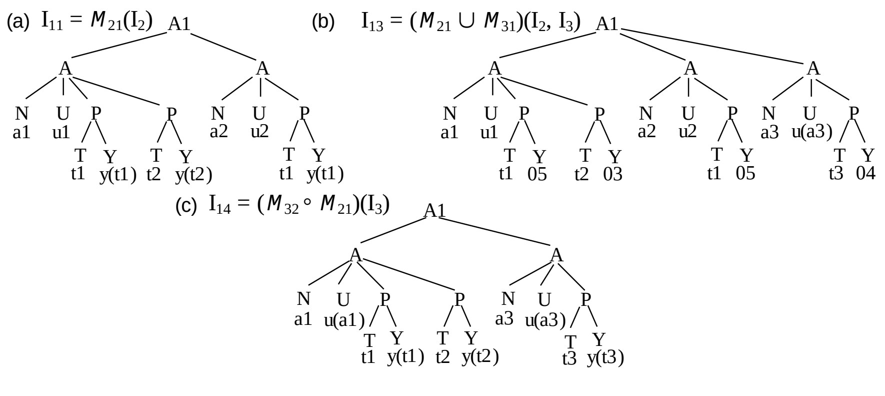


Fig. 3. Instances of schema $\mathbf{S}_1$ produced by mappings using constraints

Suppose that under $\mathbf{S}_1$ we want to merge the instances I_{11} (Fig. 3(a)) and I_3 (Fig. 1). In this process terms denoting years will be replaced with actual values (Fig. 3(b)). In this way we are able to infer the publication year of the paper written by $a2$. This information is not given explicitly neither in I_2 nor in I_3. The instances in Fig. 3(a)-(c) illustrate execution of composed mappings. In detail, we will address this issue in Section 6.

Information provided by key constraints will be used to specify how many instances (nodes) of an element type must be in the computed target instance. For example, the element of type $/A1/A$ in $\mathbf{S}_1$ is uniquely identified by the key path N. So, there are as many nodes of type $/A1/A$ as there are different values of $/A1/A/N$. In $\mathbf{S}_2$, however, elements of type $/P2/P/A$ are identified by N but only in a context determined by the element type $/P2/P$ that is identified by T. Thus, to identify $/P2/P/A$ we need a pair of values determined by paths $/P2/P/T$ and $/P2/P/A/N$. In XML Schema such keys are defined using `<xs:key>`, and a declaration within a subelement denotes that the key identification is satisfied only in the context of superelement. In Fig. 4 there is a definition of the schema $\mathbf{S}_1$ written in an extended variant of XML Schema.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="A1">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="A"/>
      </xs:sequence>
    </xs:complexType>
  </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:element ref="P" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="AKey">
      <xs:selector xpath="."/>
      <xs:field xpath="N"/>
    </xs:key>
    <xs:valdep>
      <xs:dependent xpath="N"/>
      <xs:function name="u"/>
      <xs:argument xpath="N"/>
    </xs:valdep>
  </xs:element>
  <xs:element name="P">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="T" type="xs:string"/>
        <xs:element name="Y" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="PKey">
      <xs:selector xpath="."/>
      <xs:field xpath="T"/>
    </xs:key>
    <xs:valdep>
      <xs:dependent xpath="Y"/>
      <xs:function name="y"/>
      <xs:argument xpath="T"/>
    </xs:valdep>
  </xs:element>
</xs:schema>
```

Fig. 4. XML Schema of $\mathbf{S}_1$, extended with `<xs:valdep>` declaration

5 Executable XML schema mappings

5.1 Basic ideas of mappings

From the definition in Fig. 4 we can generate the automapping $\mathcal{M}_{11}$ over $\mathbf{S}_1$ (Fig. 5), i.e. a mapping from $\mathbf{S}_1$ onto itself (numbers of lines provided here are for explanation only). It is formalized in Definition 2.

$\mathcal{M}_{11} =$ **foreach** G_{11} **where** Φ_{11} **when** C_{11} **exists** $\Delta_{11} =$

(1) **foreach** $\$y_{A1}$ $\underline{\text{in}}$ $/A1$, $\$y_A$ $\underline{\text{in}}$ $\$y_{A1}/A$, $\$y_N$ $\underline{\text{in}}$ $\$y_A/N$, $\$y_U$ $\underline{\text{in}}$ $\$y_A/U$, $\$y_P$ $\underline{\text{in}}$ $\$y_A/P$, $\$y_T$ $\underline{\text{in}}$ $\$y_P/T$, $\$y_Y$ $\underline{\text{in}}$ $\$y_P/Y$
(2) **where true**
(3) **when** $\$y_U = u(\$y_N), \$y_Y = y(\$y_T)$
exists
(4) $F_{/A1}()$ **in** $F_{()}()/A1$
(5) $F_{/A1/A}(\$y_N)$ **in** $F_{/A1}()/A$
(6) $F_{/A1/A/N}(\$y_N)$ **in** $F_{/A1/A}(\$y_N)/N$ **with** $\$y_N$
(7) $F_{/A1/A/U}(\$y_N, \$y_U)$ **in** $F_{/A1/A}(\$y_N)/U$ **with** $\$y_U$
(8) $F_{/A1/A/P}(\$y_N, \$y_T)$ **in** $F_{/A1/A}(\$y_N)/P$
(9) $F_{/A1/A/P/T}(\$y_N, \$y_T)$ **in** $F_{/A1/A/P}(\$y_N, \$y_T)/T$ **with** $\$y_T$
(10) $F_{/A1/A/P/Y}(\$y_N, \$y_T, \$y_Y)$ **in** $F_{/A1/A/P}(\$y_N, \$y_T)/Y$ **with** $\$y_Y$

Fig. 5. Automapping $\mathcal{M}_{11}$ over $\mathbf{S}_1$

(1) The clause **foreach** defines *source variables.* Every source variable ranges over a set of nodes. Variables ranging over non-leaf nodes are *auxiliary variables* whereas variables ranging over leaves are *text variables.* Note that auxiliary variables appear only in the **foreach** clause. We assume that any text variable is partially bound to text values of leaves over which it ranges. By Ω we will denote a set of all (partial) bindings for source text variables.

(2) The **where** clause restricts values of variables. The restrictions can be consequences of key references defined in the schema (see $\mathcal{M}_{33}$ in Fig. 6).

(3) Equalities in (3) reflect value dependency constraints specified in the schema. They are interpreted as follows. Let $y = f(\overline{\$x})$ be a value dependency, where $\overline{\$x}$ is a vector of (totally bound) source variables, f is the name of a text valued Skolem function, and $\$y$ is a *dependent variable* that denotes a text value in the target (e.g. $\$y_U$ and $\$y_Y$). The value of $f(\overline{\$x})$ is determined by a binding $\omega \in \Omega$ and is equivalent to the term "$f(\omega(\overline{\$x}))$" being the result of the concatenation of the name "f" and the text value $\omega(\overline{\$x})$ created from the current values of variables. We assume that we have a set Ω'_{Ω} of *dependent bindings* such that for any value dependency $\$y = f(\overline{\$x})$ and for any $\omega \in \Omega$ there exists $\omega'_{\omega} \in \Omega'_{\Omega}$ such that $\omega'_{\omega}(\$y) :=$ "$f(\omega(\overline{\$x}))$". If $\omega(\$y)$ is defined, then there are two bindings for $\$y$: $\omega(\$y)$ and $\omega'_{\omega}(\$y)$, otherwise the only binding for $\$y$ is $\omega'_{\omega}(\$y)$. In some cases, dependent bindings can be used to infer missing bindings (i.e. values of variables) by applying the following inference rule:

$$\omega'_{\omega_1}(\$y_1) = \omega'_{\omega_2}(\$y_2) \Rightarrow \omega_1(\$y_1) = \omega_2(\$y_2) \quad (1)$$

In this way we can obtain the value of $\omega_1(\$y_1)$(see Example 4).

(4) Two new nodes are created, the root r and the node n of the outermost element of type $/A1$, as results of Skolem functions $F_{()}()$ and $F_{/A1}()$, respectively. The node n is a child of type $A1$ of r.

(5) A new node n' for any distinct value of $\$y_N$ is created. Each such node has the type $/A1/A$ and is a child of type A of the node n created by $F_{/A1}()$ in (4).

(6) For any distinct value of $\$y_N$ a new node n'' of type $/A1/A/N$ is created. Each such node is a child of type N of the node created by invocation of $F_{/A1/A}(\$y_N)$ in (5) for the same value of $\$y_N$. Because n'' is a leaf, it obtains the text value equal to the current value of $\$y_N$.

(7) Analogously for the rest of the specification (7-10).

5.2 Capturing key constraints by automappings

In a specification of automapping, Skolem functions and their arguments play a crucial role. We assume that:

- for any (rooted) path P in the schema there is exactly one Skolem function, $F_P(...)$, where F_P is the name of the Skolem function,
- arguments of a Skolem function $F_P(...)$ are determined by key paths defined for the element of type P in the schema.

In $\mathbf{S}_1$ there is exactly one root and one outermost element, so the corresponding Skolem functions have empty lists of arguments. Element of type $/A1/A$ has a key path N. Each its subelement inherits this key path and additionally has its local (relative) key paths. The local key paths for non-leaf elements are defined in the schema. The local key path for a leaf element is, by default, this leaf element itself. Thus, for $\mathbf{S}_1$ we have the following key paths: N for $/A1/A$ and $/A1/A/N$; (N,T) for $/A1/A/P$ and $/A1/A/P/T$; and (N,T,Y) for $/A1/A/P/Y$. Text values of these key paths are bound to variables and are used as arguments of Skolem functions.

In definition of $\mathbf{S}_3$ (Fig. 6), the schema specifies the *key* and *keyref* relationships between the K child element of the P element (the *referenced key*) and the R child element of the A element (the *foreign key*).

In the automapping specification over $\mathbf{S}_3$, key references are captured by the equality $\$z_R = \$z_K$ in the **where** clause (Fig. 6).

```
...
<xs:element name="A">
  <xs:complexType>
  ...
  </xs:complexType>...
  <xs:keyref name="AKeyref"
           refer="PKey">
    <xs:selector xpath="."/>
    <xs:field xpath="R"/>
  </xs:keyref>
</xs:element>
```

```
<xs:element name="P">
  <xs:complexType>
  ...
  </xs:complexType>
  <xs:key name="PKey">
    <xs:selector xpath="."/>
    <xs:field xpath="K"/>
  </xs:key>
  <xs:valdep>
    <xs:dependent name="Y"/>
    <xs:function name="y"/>
    <xs:argument xpath="T"/>
  </xs:valdep>   ...
</xs:element>
...
```

```
$\mathcal{M}_{33} =$ foreach $\$z_{D3}$ in $/D3$, $\$z_A$ in $\$z_{D3}/A$, $\$z_N$ in $\$z_A/N$, $\$z_R$ in $\$z_A/R$,
               $\$z_P$ in $\$z_{D3}/P$, $\$z_K$ in $\$z_P/K$, $\$z_T$ in $\$z_P/T$,
               $\$z_Y$ in $\$z_P/Y$, $\$z_C$ in $\$z_P/C$
    where $\$z_R = \$z_K$
    when $\$z_K = k(\$z_N, \$z_T), \$z_Y = y(\$z_T), \$z_C = c(\$z_T)$
    exists
        $F_{/D3}()$ in $F_{()}()/D3$
        $F_{/D3/A}(\$z_N)$ in $F_{/D3}()/A$
        $F_{/D3/A/N}(\$z_N)$ in $F_{/D3/A}(\$z_N)/N$ with $\$z_N$
        $F_{/D3/A/R}(\$z_N, \$z_R)$ in $F_{/D3/A}(\$z_N)/R$ with $\$z_R$
        $F_{/D3/P}(\$z_K)$ in $F_{/D3}()/P$
        $F_{/D3/P/K}(\$z_K)$ in $F_{/D3/P}(\$z_K)/K$ with $\$z_K$
        $F_{/D3/P/T}(\$z_K, \$z_T)$ in $F_{/D3/P}(\$z_K)/T$ with $\$z_T$
        $F_{/D3/P/Y}(\$z_K, \$z_Y)$ in $F_{/D3/P}(\$z_K)/Y$ with $\$z_Y$
        $F_{/D3/P/C}(\$z_K, \$z_C)$ in $F_{/D3/P}(\$z_K)/C$ with $\$z_C$
```

Fig. 6. Fragment of XML Schema defining $\mathbf{S}_3$ and the automapping $\mathcal{M}_{33}$ over $\mathbf{S}_3$

5.3 Syntax and semantics for mappings

In general, there are two vectors of variables $\overline{\$x}$ and $\overline{\$y}$ in a mapping $\mathcal{M}$. Variables from $\overline{\$x}$ are bound in a source by means of the **foreach** clause, and variables from $\overline{\$y}$ are bound to terms in the **when** clause as a consequence of value dependency constraints. The part **foreach/where/when** of a mapping $\mathcal{M}(\overline{\$x};\overline{\$y})$ determines a partially ordered set $(\Omega, \leq)$ of bindings of $\mathcal{M}$'s variables. For example, in the mapping $\mathcal{M}_{21}$ (Fig. 7) for two bindings $\omega_1, \omega_2 \in \Omega$ over I_2, where $\omega_1 = (\$x_T \to t_1, \$x_N \to a_1, \$x_U \to u_1, \$y_Y \to y(t_1))$ and $\omega_2 = (\$x_T \to t_1, \$x_N \to a_2, \$x_U \to u_2, \$y_Y \to y(t_2))$, we have $\omega_1 < \omega_2$, because the tuple of leaf nodes providing values for ω_1 precedes the tuple of leaf nodes providing values for ω_2. Bindings from Ω are used in the **exists** part to produce the result target instance. The ordering imposed in Ω by a source instance should be preserved in the target instance.

If the **foreach/where** clause is defined over $\mathbf{S}_2$, while the **when/exists** concerns $\mathbf{S}_1$, then we deal with a mapping $\mathcal{M}_{21}$ from $\mathbf{S}_2$ into $\mathbf{S}_1$. Then, after the given replacement of variables (the result of the replacement $\phi[\$y \to \$x]$ is the expression created from ϕ by replacing all occurrences of $\$y$ with $\$x$), we obtain:

$$
\begin{array}{ll}
\mathcal{M}_{21} = & \textbf{foreach } \$x_{P2} \ \underline{\text{in}} \ /P2, \ \$x_P \ \underline{\text{in}} \ \$x_{P2}/P, \ \$x_T \ \underline{\text{in}} \ \$x_P/T, \\
& \quad \$x_A \ \underline{\text{in}} \ \$x_P/A, \ \$x_N \ \underline{\text{in}} \ \$x_A/N, \ \$x_U \ \underline{\text{in}} \ \$x_A/U \\
& \textbf{where true} \\
& \textbf{when } C_{11}(\$y_N, \$y_U, \$y_T, \$y_Y)[\$y_N \to \$x_N, \$y_U \to \$x_U, \$y_T \to \$x_T] \\
& \textbf{exists } \Delta_{11}(\$y_N, \$y_U, \$y_T, \$y_Y)[\$y_N \to \$x_N, \$y_U \to \$x_U, \$y_T \to \$x_T]
\end{array}
$$

Fig. 7. Mapping $\mathcal{M}_{21}$ from $\mathbf{S}_2$ into $\mathbf{S}_1$

Thus, the **when** clause of $\mathcal{M}_{21}$ is equal to $\$x_U = u(\$x_N), \$y_Y = y(\$x_T)$. There is no replacement for $\$y_Y$, so its value must be set as the current value of the term $y(\$x_T)$, according to the inference rule (1) proposed in 5.1. We set it as the term $y(t)$, where t is the current value of $\$x_T$ (see Fig. 3(a)). It is a form of *Skolemization*.

Observe that a mapping specification in XDMap conforms to the general form of *source-to-target generating dependencies* [2, 6, 13]:

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

where $G(\overline{\$x})$ and $\Phi(\overline{\$x})$ are conjunctions of atomic formulas over a source, and $C(\overline{\$x};\overline{\$y})$ and $\Delta(\overline{\$x};\overline{\$y})$ are conjunctions of atomic formulas over a target.

Definition 1. *An executable schema mapping in XDMap (or mapping for short) between a source schema* $\mathbf{S}$ *and a target schema* $\mathbf{T}$ *is a sequence* $\mathcal{M} ::= (M, ..., M)$ *of mapping rules between* $\mathbf{S}$ *and* $\mathbf{T}$*, where:*

$$
\begin{array}{ll}
M = (G, \Phi, C, \Delta)(\overline{\$x};\overline{\$y}) := & \textbf{foreach } G(\overline{\$x}) \\
& \textbf{where } \Phi(\overline{\$x}) \\
& \textbf{when } C(\overline{\$x};\overline{\$y}) \\
& \textbf{exists } F_{P/l}(\overline{\$x'};\overline{\$y'}) \textbf{ in } F_P(\overline{\$x''};\overline{\$y''})/l[\textbf{ with } \$z\ ]
\end{array}
$$

– *G is a list of variable definitions over a source schema;*
– *Φ is a conjunction of atomic conditions: $\$x = \$x'$;*
– *C is a list of target value dependency constraints: $\$x = f(\overline{\$x})$ or $\$y = f(\overline{\$x})$, $\$x \in \overline{\$x}$, $\$y \in \overline{\$y}$;*
– *$F_P(\overline{\$x}; \overline{\$y})$ is a Skolem term, where P is a rooted path in a target schema;*
– *$(\overline{\$x'}; \overline{\$y'}) \subseteq (\overline{\$x}; \overline{\$y})$, $(\overline{\$x''}; \overline{\$y''}) \subseteq (\overline{\$x'}; \overline{\$y'})$, $\$z \in (\overline{\$x'}; \overline{\$y'})$.* $\square$

Semantics for XDMap is defined as follows:

Definition 2. *Let $\mathcal{M} = (G, \Phi, C, \Delta)(\overline{\$x}; \overline{\$y})$ be a mapping, and $(\Omega, \leq)$ be a partially ordered set of bindings of variables $(\overline{\$x}; \overline{\$y})$ determined by (G, Φ, C). A target instance J of a target schema $\mathbf{T}$ is then obtained as follows:*

1. *$F_{()}()$ – the root of J.*
2. *$F_P(\overline{\$x'}; \overline{\$y'})(\omega) = n$ – a node of type P.*
3. *If $F_{P/l}(\overline{\$x'}; \overline{\$y'})(\omega) = n$, $F_P(\overline{\$x''}; \overline{\$y''})(\omega) = n'$, then n is a child of type l of n'.*
4. *Let $F_{P/l}(\overline{\$x'}; \overline{\$y'})(\omega_1) = n_1$, $F_{P/l}(\overline{\$x'}; \overline{\$y'})(\omega_2) = n_2$, and $\omega_1 \leq \omega_2$. Then $n_1 \leq n_2$ in the document order.*
5. *If $F_{P/l}(\overline{\$x'}; \overline{\$y'})(\omega) = n$ is a leaf, then the text value of n is $\omega(\$z)$.*

6 Operations on mappings

Mappings can be combined by means of some operators giving a result that in turn is a mapping. We define the following operations: *Match*, *Compose*, and *Merge*. First, we have to define a *correspondence* between paths of different schemas. Establishing the correspondence is a crucial task in definition of data mappings [17].

Definition 3. *Let $\mathcal{P}$ and $\mathcal{P}'$ be sets of paths from schemas $\mathbf{S}$ and $\mathbf{S}'$, respectively. A correspondence from $\mathbf{S}$ into $\mathbf{S}'$ is a partial function $\sigma : \mathcal{P} \to \mathcal{P}'$ which maps a path $P \in \mathcal{P}$ on a path $P' = \sigma(P) \in \mathcal{P}'$.* $\square$

Example 1. Correspondence σ_{12} from $\mathbf{S}_1$ to $\mathbf{S}_2$, and σ_{23} from $\mathbf{S}_2$ to $\mathbf{S}_3$ (Fig. 1) are:

$$\begin{array}{ll}
\sigma_{12}(/A1/A/N) = /P2/P/A/N & \sigma_{23}(/P2/P/T) = /D3/P/T \\
\sigma_{12}(/A1/A/U) = /P2/P/A/U & \sigma_{23}(/P2/P/A/N) = /D3/A/A \\
\sigma_{12}(/A1/A/P/T) = /P2/P/T &
\end{array}$$

6.1 *Match* operator

The *Match* operator was proposed in [11] as an operator to create a mapping between two schemas (modes). In our approach each schema is represented by automappings, thus *Match* is defined on two automappings and returns a mapping between schemas over which these automappings are defined. Because *Match* is in fact a special kind of composition, we will denote it, like the *Compose* operator (see p. 6.2), by $\circ$, i.e. $Match(\mathcal{M}_s, \mathcal{M}_t)$ will be abbreviated by $\mathcal{M}_s \circ \mathcal{M}_t$, where

$\mathcal{M}_s$ and $\mathcal{M}_t$ are automappings over source and target schemas, respectively. Then $\mathcal{M}_{st} = \mathcal{M}_s \circ \mathcal{M}_t$ is a mapping from the source schema into the target schema.

Definition 4. *Let $\mathcal{M}_s = (G_s, \Phi_s, C_s, \Delta_s)(\overline{\$x_s}; \overline{\$y_s})$ be an automapping over $\mathbf{S}$, $\mathcal{M}_t = (G_t, \Phi_t, C_t, \Delta_t)(\overline{\$x_t}; \overline{\$y_t})$ be an automapping over $\mathbf{T}_t$, and σ be a correspondence between $\mathbf{T}$ and $\mathbf{S}$. Then the $Match_\sigma(\mathcal{M}_s, \mathcal{M}_t)$ is the mapping*

$$\mathcal{M}_s \circ_\sigma \mathcal{M}_t = (G_s, \Phi_s, C_t, \Delta_t)[\overline{\$x_t} \to \sigma(\overline{\$x_t})](\overline{\$x}; \overline{\$y}), \qquad (2)$$

where the result of the replacement $[\overline{\$x_t} \to \sigma(\overline{\$x_t})]$ is defined as follows:

- *any occurrence of a variable $\$x_t \in \overline{\$x_t}$ of type P in $(G_s, \Phi_s, C_t, \Delta_t)$ is replaced by a variable $\$x_s \in \overline{\$x_s}$ of type $\sigma(P)$, $\$x_s$ is the replacing variable; and $\overline{\$x}$ is a tuple of all replacing variables and all variables occurring in Φ_s;*
- *$\overline{\$y} \subseteq (\overline{\$x_t}; \overline{\$y_t})$ and consists of all variables which have not been replaced; and all unnecessary variable definitions are removed from G_s.* □

Example 2. The mapping $\mathcal{M}_{21}$ (Fig. 7) is the result of matching from $\mathcal{M}_{22}$ to $\mathcal{M}_{11}$ using the correspondence σ_{12} from $\mathbf{S}_1$ to $\mathbf{S}_2$ (see Example 1), i.e.

$\mathcal{M}_{21}(\$x_N, \$x_U, \$x_T; \$y_Y) = \mathcal{M}_{22}(\$x_T, \$x_N, \$x_U) \circ_{\sigma_{12}} \mathcal{M}_{11}(\$y_N, \$y_U, \$y_T, \$y_Y)$.

Similarly,

$\mathcal{M}_{32}(\$z_T, \$z_N, \$z_R, \$z_K; \$v_U) =$
$= \mathcal{M}_{33}(\$z_N, \$z_R, \$z_K, \$z_T, \$z_Y, \$z_C) \circ_{\sigma_{23}} \mathcal{M}_{22}(\$v_T, \$v_N, \$v_U)$. □

6.2 *Compose* operator

The *Compose* operator combines two successive mappings into one.

Definition 5. *Let $\mathcal{M}_{12}$ and $\mathcal{M}_{23}$ be mappings from $\mathbf{S}_1$ into $\mathbf{S}_2$ and from $\mathbf{S}_2$ into $\mathbf{S}_3$, respectively. Let σ_{21} and σ_{32} be correspondences from $\mathbf{S}_2$ into $\mathbf{S}_1$ and from $\mathbf{S}_3$ into $\mathbf{S}_2$. Let $\mathcal{M}_{11}$ and $\mathcal{M}_{33}$ be automappings over $\mathbf{S}_1$ and $\mathbf{S}_3$, respectively, and $\sigma = \sigma_{32} \circ \sigma_{21}$ be the correspondence from $\mathbf{S}_3$ into $\mathbf{S}_2$ obtained as the result of composition of correspondences σ_{32} and σ_{21}. Then*

$$Compose_\sigma(\mathcal{M}_{12}, \mathcal{M}_{23}) = \mathcal{M}_{11} \circ_\sigma \mathcal{M}_{33} \qquad (3)$$

is a mapping from $\mathbf{S}_1$ to $\mathbf{S}_3$, and $\mathcal{M}_{11} \circ_\sigma \mathcal{M}_{33}$ is defined by (2).

Example 3. It is easily to show that:

$\mathcal{M}_{321}(\$z_T, \$z_N, \$z_R, \$z_K; \$y_U, \$y_Y) = Compose_{(\sigma_{12} \circ \sigma_{23})}(\mathcal{M}_{32}, \mathcal{M}_{21}) =$
$\mathcal{M}_{33}(\$z_N, \$z_R, \$z_K, \$z_T, \$z_Y, \$z_C) \circ_{(\sigma_{12} \circ \sigma_{23})} \mathcal{M}_{11}(\$y_N, \$y_U, \$y_T, \$y_Y) =$

```
= foreach $G_{33}(\$z_N, \$z_R, \$z_K, \$z_T)$
  where $\$z_R = \$z_K$
  when $\$y_U = u(\$z_N), \$y_Y = y(\$z_T)$
  exists
      $F_{/A1}()$ in $F_{()}()/A1$
      $F_{/A1/A}(\$z_N)$ in $F_{/A1}()/A$
      $F_{/A1/A/N}(\$z_N)$ in $F_{/A1/A}(\$z_N)/N$ with $\$z_N$
      $F_{/A1/A/U}(\$z_N, \$y_U)$ in $F_{/A1/A}(\$z_N)/U$ with $\$y_U$
      $F_{/A1/A/P}(\$z_N, \$z_T)$ in $F_{/A1/A}(\$z_N)/P$
      $F_{/A1/A/P/T}(\$z_N, \$z_T)$ in $F_{/A1/A/P}(\$z_N, \$z_T)/T$ with $\$z_T$
      $F_{/A1/A/P/Y}(\$z_N, \$z_T, \$y_Y)$ in $F_{/A1/A/P}(\$z_N, \$z_T)/Y$ with $\$y_Y$
```

$\mathcal{M}_{321}$ has two variables, $\$y_U$ and $\$y_Y$, which are not bound in the source. Instead, they are bound in the **when** clause to target terms $u(\$z_N)$ and $y(\$z_T)$, respectively. An instance of the mapping is given in Fig. 3(c). In the final result all term-valued leaves may be either removed, replaced with nulls, or left as they are (they may be resolved and replaced with actual values in next mappings (e.g. by *Merge*) as in Fig. 3(a)-(b)).

6.3 *Merge* operator

Definition 6. *Let* $\mathcal{M}_1 = (G_1, \Phi_1, C_1, \Delta_1)(\overline{\$x_1}; \overline{\$y_1})$ *and* $\mathcal{M}_2 = (G_2, \Phi_2, C_2, \Delta_2)(\overline{\$x_2}; \overline{\$y_2})$, *where* $(\overline{\$x_1}; \overline{\$y_1})$ *and* $(\overline{\$x_2}; \overline{\$y_2})$ *are disjoint, be mappings from* $\mathbf{S}_1$ *and* $\mathbf{S}_2$, *respectively, into* $\mathbf{S}_3$. *Then merging of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *is the mapping defined as follows:*

$$\mathcal{M}_1 \cup \mathcal{M}_2 = (G_1 \cup G_2, \Phi_1 \cup \Phi_2, C_1 \cup C_2, \Delta_1 \cup \Delta_2)(\overline{\$x_1} \cup \overline{\$x_2}; \overline{\$y_1} \cup \overline{\$y_2}).$$

If mappings $\mathcal{M}_1$ and $\mathcal{M}_2$ are mappings from $\mathbf{S}_1$ and $\mathbf{S}_2$, respectively, into $\mathbf{S}_3$ then $\mathcal{M}_1 \cup \mathcal{M}_2$ is a mapping that *merges* $\mathbf{S}_1$ and $\mathbf{S}_2$ under $\mathbf{S}_3$.

Example 4. Let
$\mathcal{M}_{21} = (G_{21}(\$x_T, \$x_N, \$x_U), \{\mathbf{true}\},$
$\{x_U = u(\$x_N), \$y_Y = y(\$x_T)\}, \Delta_{21}(\$x_T, \$x_N, \$x_U; \$y_Y))$
be a mapping from $\mathbf{S}_2$ into $\mathbf{S}_1$ (Example 2), and
$\mathcal{M}_{31} = (G_{31}(\$z_N, \$z_R, \$z_K, \$z_T, \$z_Y), \{\$z_R = \$z_K\},$
$\{\$v_U = u(\$z_N), \$z_Y = y(\$z_T)\}, \Delta_{31}(\$z_N, \$z_T, \$z_Y; \$v_U))$
be a mapping from $\mathbf{S}_3$ into $\mathbf{S}_1$.

The merge $\mathcal{M}_{21} \cup \mathcal{M}_{31}$ is a mapping consisting of all mapping rules from $\mathcal{M}_{21}$ and all mapping rules from $\mathcal{M}_{31}$. Below, we show only these rules that involve variables from constraints in the **when** clause.

```
$\mathcal{M}_{21} \cup \mathcal{M}_{31}$ =
    **foreach** $G_{21}(\$x_T, \$x_N, \$x_U), G_{31}(\$z_N, \$z_R, \$z_K, \$z_T, \$z_Y)$
    **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**  ...
                $F_{/A1/A/U}(\$x_N, \$x_U)$ **in** $F_{/A1/A}(\$x_N)/U$ **with** $\$x_U$
                $F_{/A1/A/P/Y}(\$x_N, \$x_T, \$y_Y)$ **in** $F_{/A1/A/P}(\$x_N, \$x_T)/Y$ **with** $\$y_Y$
                $F_{/A1/A/U}(\$z_N, \$v_U)$ **in** $F_{/A1/A}(\$z_N)/U$ **with** $\$v_U$
                $F_{/A1/A/P/Y}(\$z_N, \$z_T, \$z_Y)$ **in** $F_{/A1/A/P}(\$z_N, \$z_T)/Y$ **with** $\$z_Y$
                ...
```

For variables occurring in G_{21} and in G_{31}, the **foreach** clause defines a set Ω of bindings. Term values of dependent variables, i.e. of $\$x_U$, $\$y_Y$, $\$z_Y$, and $\$v_U$, are defined in the **when** clause and are represented by Ω'. Missing values in Ω (e.g. for variables $\$y_Y$ and $\$v_U$) are replaced by appropriate term values from Ω', i.e. $\omega(\$y) := \omega'_{\omega}(\$y)$.

Before resolving: $\Omega := \Omega_{21} \cup \Omega_{31}$

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

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

Next, in the *resolving process* we try to resolve term values in Ω. The resolving process is based on the rule (1) discussed in Subsection 5.1.

After resolving: $\Omega := Resolve(\Omega_{21} \cup \Omega_{31})$

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	$u(a3)$

Execution of the mapping $\mathcal{M}_{21} \cup \mathcal{M}_{31}$ is illustrated in Fig. 3. Fig. 3(a) shows the result produced by the part corresponding to $\mathcal{M}_{21}$, and Fig. 3(b) is the final result. Note, that the term $u(a_3)$ cannot be resolved. $\square$

7 Conclusion

We have described a novel approach to XML schema mapping specification and operations over schema mappings. We discussed how automappings may be generated using key constraints [4, 18], keyref constraints [18], and some value dependency constraints defined in XML Schema. Constraints on values can be used to infer some missing data. Mappings between two schemas can be generated automatically from their automappings and correspondences between paths of these two schemas. Automappings represent schemas, so operations over schemas and mappings can be defined and performed in a uniform way. We propose some algebraic operations over mappings. The syntax and semantics for the mapping language XDMap are defined and discussed. 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) [14, 15].

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. Fagin, R., Kolaitis, P. G., Popa, L.: Data exchange: getting to the core., *ACM Trans. Database Syst.*, **30**(1), 2005, 174–210.
6. Fagin, R., Kolaitis, P. G., Popa, L., Tan, W. C.: Composing Schema Mappings: Second-Order Dependencies to the Rescue., *PODS*, 2004, 83–94.
7. 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.
8. Hull, R., Yoshikawa, M.: ILOG: Declarative Creation and Manipulation of Object Identifiers., *VLDB*, 1990, 455–468.
9. Kuikka, E., Leinonen, P., Penttonen, M.: Towards automating of document structure transformations., *ACM Symposium on Document Engineering*, 2002, 103–110.
10. Madhavan, J., Halevy, A. Y.: Composing Mappings Among Data Sources., *VLDB*, 2003, 572–583.
11. Melnik, S., Bernstein, P. A., Halevy, A. Y., Rahm, E.: Supporting Executable Mappings in Model Management., *SIGMOD Conference*, 2005, 167–178.
12. Melnik, S., Rahm, E., Bernstein, P. A.: Rondo: A Programming Platform for Generic Model Management., *SIGMOD Conference*, 2003, 193–204.
13. Nash, A., Bernstein, P. A., Melnik, S.: Composition of Mappings Given by Embedded Dependencies., *PODS*, 2005.
14. Pankowski, T.: Specifying Schema Mappings for Query Reformulation in Data Integration Systems, 3rd *Atlantic Web Intelligence Conference - AWIC'2005,* Lecture Notes in Computer Science 3528, Springer-Verlag, 2005, 361–365.
15. Pankowski, T.: Integration of XML Data in Peer-To-Peer E-commerce Applications, 5th *IFIP Conference I3E'2005*, Springer, New York, 2005, 481–496.
16. Popa, L., Velegrakis, Y., Miller, R. J., Hernández, M. A., Fagin, R.: Translating Web Data., *VLDB*, 2002, 598–609.
17. Rahm, E., Bernstein, P. A.: A survey of approaches to automatic schema matching, *The VLDB Journal*, **10**(4), 2001, 334–350.
18. XML Schema Part 1: Structures 2d Edition: 2004. www.w3.org/TR/xmlschema-1
19. Yu, C., Popa, L.: Constraint-Based XML Query Rewriting For Data Integration., *SIGMOD Conference*, 2004, 371–382.
20. Yu, C., Popa, L.: Semantic Adaptation of Schema Mappings when Schemas Evolve., *VLDB*, 2005, 1006–1017.