HAL open science

Automating XML document transformations with structuring mapping result

Amar Zerdazi, Myriam Lamolle

To cite this version:

Amar Zerdazi, Myriam Lamolle. Automating XML document transformations with structuring mapping result. CSREA EEE, 2007, Las Vegas, United States. hal-03580931

HAL Id: hal-03580931

https://hal.science/hal-03580931v1

Submitted on 18 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Automating XML document transformations with structuring mapping result

Amar ZERDAZI, Myriam LAMOLLE

LINC – Laboratory of Paris VIII
140, rue de la Nouvelle France 93000 - Montreuil, France
Tel.: +33148703464 – Fax: +33148703467
{a.zerdazi, m.lamolle}@iut.univ-paris8.fr

Abstract. This paper describes a transformation of XML documents from one structure to another could be generated on previously established mappings introduced in our earlier work. That work characterized certain general conditions under which a semi-automatic transformation is possible. The system generates a transformation between two structures of the same document class. We begin by introducing the chosen structure of the mapping result. Then we describe how we generate automatically a transformation script based on such structured mapping result.

Keywords: script XSLT, structuring mapping transformation, XML document.

1 Introduction

Today's web-based applications and web services publish their data using XML, as this helps interoperability with other applications and services. The heterogeneity of XML data has led to recent research in schema matching, schema transformation, and schema integration for XML. The development of algorithms that automate these tasks is essential to many domains. As the number of applications that utilize heterogeneous XML documents grows, the importance of XML documents transformations increases greatly. Currently, such transformations are manually using specific languages such as XSLT [11]. XSLT, a recommendation of the World Wide Web Consortium, is a language, itself written in XML, with powerful computing capability encoding transformation of XML documents. An XSLT program is a set of template rules, each of which has two parts: a pattern that is matched against nodes in the source document and a template that can be instantiated to form the result document. The usual procedure to write such stylesheet requires an analysis of both the semantics and the structures of the source and target XML files to discover similarities between them.

The goal of this paper is to propose an approach for automating the transformation of XML documents. We focus on two fundamental problems. First, we address the problem of how to automatic the identification of semantic relationships between XML schemas. To this, we have proposed in previous work a matching framework that incorporates several matching criteria and detects mappings as well as complex mappings. Second, given such mappings, we need to perform the actual transformation of XML documents from a source schema to a different, yet related target schema. The paper is organized in the following way. In section 2, we summarize some examples of recent schema matching algorithms that incorporate XML structural matching. Section 3 gives a brief overview of our formal data model for XML schema and a set of a set of primitive transformation operations. Section 4 presents the core of this paper. We essentially proceed in two steps, structuring mapping result and generating XSLT transformations (figure 1). We give an example that illustrates our approach. Finally, Section 5 concludes the paper.

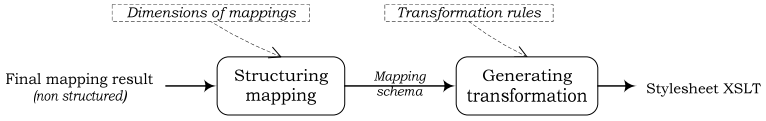

FIG 1 – Process of automating transformations.

2 State of the art

Many solutions have been proposed in order to automate XML documents transformations. Approaches can be distinguished along the following three dimensions: schema translation, schema matching, and XML document restructuring.

Schema translation. Clio [7] translates the data source schemas, XML or relational, into an internal representation. After the mappings between the source and the target schemas have been semiautomatically derived, the target schema is materialized from the data of the sources, using a set of rules and the mappings. DIXSE [9] transforms the DTD specifications of a set of source XML documents into an internal conceptual representation, using some heuristics to capture semantics and further input from domain experts. ARTEMIS [1], [2] supports the analysis and reconciliation of sets of heterogeneous relational schemas by measuring the similarity of element names, data types, and structures. The LSD system [3] uses machine-learning techniques to match a new data source against a previously defined global schema. LSD is based on the combination of several match result obtained by independent learners.

Schema matching. Cupid [4] is a hybrid approach that considers both tag names and hierarchical structures of schema. The similarity between an element of the first schema and an element of the second schema relies on the similarity of their components hereby emphasizing the name and data type similarities present at the finest granularity level. SF (Similarity Flooding) [6] is a hybrid approach based on the ideas of similarity propagation. The SF algorithm is implemented as part of a generic schema manipulation tool that supports, in addition to structural SF matcher, a name matcher, schema converters and a number of filters of choosing the best match candidates from the list of ranked map pairs returned by the SF algorithm. [10] offers a novel integration approach that uses semi-automatic schema matching to produce source-to-target mappings. A recent survey of automatic schema matching [8] classifies approaches respecting to the schema matching.

3 The data model

As we already mention in section 2, up to now few existent XML schema matching algorithms focus on structural matching exploiting all W3C XML schemas [12] features. In this section, we propose an abstract model that serves as a foundation to represent conceptually W3C XML schemas and potentially other schema languages. We model XML schemas as a ***directed labeled graph*** with constraint sets; so-called ***schema graph***. Schema graph consists of series of nodes that are connected to each other through directed labeled links. In addition, constraints can be defined over nodes and links. In [13], we detail the proposed model for XML schemas in order to define a formal framework for solving matching problem. Figure 2 illustrates a schema graph example.

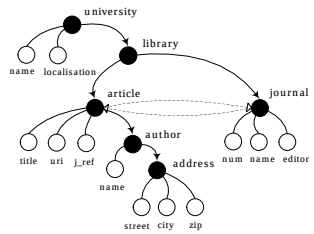

FIG 2 – A schema graph example.

Transformation operations. Our mapping algebra concerns a set of transformation operations, as listed below:

- **Rename:** *t=renameás*ñ, generates a construction[1] that is the same as a construction *s*, but with a different name *t*. For example, *editor=renameápublisher*ñ.
- **Merge:** *t=mergeás*$_1$,..*s*$_i$ñ, generates a construction *t* whose value is obtained by concatenating *s*$_1$,..*s*$_i$ in the case of strings. For example, *address=mergeástreet, city, zip*ñ.
- **Split***: (t*$_1$,..*t*$_i$*)=splitás*ñ, where *t*$_1$,..*t*$_i$ are obtained by splitting a construction *s* respecting to a separation criterion. For example, *(first-name, last-name)=splitánameñ.*
- **Parity:** *t=parityás*ñ, generates a construction *t* which has the same content and label as *s*. for example, *author=parityáauthor*ñ.

4 Structuring mapping result

4.1 Mapping structure

The role of the mapping result is so semantically relate facts from the source and target schemas by encapsulating all necessary information to transform instances of one source schema to instances of one target schema. The nature of mapping result may be understood by considering different dimensions, each describing one particular aspect. We have been inspired by the only work done in structuring mapping result [5], where authors focus on mapping distributed ontologies. We restrict ourselves to the following four *dimensions of mapping* result:

- ***Entity dimension****:* Specifies schema entities involved in a mapping element.
- ***Cardinality dimension****:* This dimension determines the cardinality of a mapping element ranging from direct mapping *(1:1)* to complex mapping *(m:n)*. However as is [5], we have found that in most cases *m:n* mappings are not common, thus we limit ourselves to *1:n* and *m:1* mappings. Even when *m:n* mappings are encountered, often they may be decomposed into *m 1:n* mapping elements.
- ***Transformation dimension****:* This dimension reflects how instances of the source are transformed during the mapping process. Transformation dimension include the identification operations.
- ***Structural dimension****:* This dimension reflects the way how elementary mapping elements may be combined into more complex mapping elements. Currently, we distinguish the structural relation between mapping elements *(composition)* specifies that a mapping is composed of other mapping elements.

To actually relate a given source and target schema graphs, the mapping process generates an instance of the mapping schema containing se set of mapping elements each of which encapsulates all information needed to

[1] Construction refers to nodes and edges in the schema graph.

transform instances of source nodes into instances of target nodes. Based on the four dimensions described above, a mapping result is described as a sequence of mapping elements each of which has a:

- A unique identifier, *map_id.*
- A type of the mapping element, *map_typ*, specified by his cardinality.
- A set of source entities involved in the mapping element, via *<source>* element.
- A set of target entities involved in the mapping element, via *<target>* element.
- A transformation element *(<transformation>)* including a transformation operation (identified in section 3).
- A set of *<child_mappings>* elements allowing the current mapping element to aggregate any number of mapping elements. Those mapping elements are then called one by one and processed in the context of the former.

4.2 Generation of mapping structure

Base on the established mapping between source and target schema [14], a mapping generator generates a structured mapping conforming to the structure previously introduced in section 4.1. For each matching node pair, the mapping generator traverses the target schema graph in the depth first manner and generates a new mapping element. The following describes how mapping elements are generated based on the target node:

1. If the node is the first matchable encountered node, then generate a top level mapping element which will serve as the entry point for the translation program.
2. For each node that is mapped, create a new mapping element.
    - Assign a *map_id* for the created mapping element[2].
    - Check the mapping rules, if the node is involved in a direct match then generate a *[1:1] (map_typ=one@one)* mapping element, else if it involves to a complex match then generate either a *[1:m] (map_typ=one@many)* or an type *[m:1] (map_typ=many@one)* mapping element.
    - Assign to the source element *(<source>)* the actual source node.
    - Assign to the target element *(<target>)* the actual target node.
    - Assign to the transformation element *(<transformation>)* the operation to the discovered operation in the mapping rule.
3. If the mapped node *n* is a complex node then:
    - For each set of edges starting at *n* and having an order composition node create a *child_mappings* elements for each matchable node connected to *n.*

4.3 Example of mapping result structuring

Let us consider the example of figure 3, where the schema graph in figure 3(a) represents the source schema and the schema graph in figure 3(b) the source target. The goal of this example is to specify a mapping between the source and target schemas, using the developed *mapping schema*. A mapping result structure represented according to the mapping schema tends to arrange mapping elements in hierarchical way (figure 4). First, the mapping must define the two schemas being mapped. Additionally, one may specify top-level mapping elements, which serve as entry points for the translation. In this case the mapping generator starts specifying that the mapping element between source node *laboratory* and target node *laboratory* is the top-level mapping element (line 1 to 4). A new mapping element is then created to describe the relation between source node *laboratory* and target node *laboratory* (line 5). Since the target node *laboratory* is a complex node described as a sequence of three children elements *name*, *address* and *researcher*, the mapping generator adjust the above mapping element bay inserting three *child_mappings* elements (line 9, 10 and 11**)** in order to

[2] Mapping elements *map_id*s are generated in an incremental automatic manner.

prepare the processing of child nodes. For each child node, a new mapping element is created. Line *13* to *18* describes the mapping between source node *name* and target node *name*, while lines *19* to *26* describes the complex mapping between source node *location* and target node *address*.

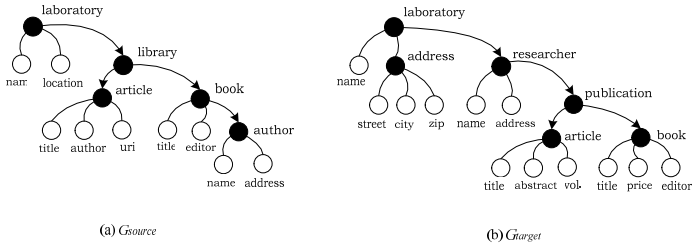

FIG 3 – Source and target schema graphs.

```
1.<mapping_structure>
2.        <source_schema  name="laboratory1.xsd"/>
3.        <cible_schema  name="laboratory2.xsd"/>
4.        <child_mappings map_id="m1"/>
5.        <mapping_element  map_id="m1"  map_typ= "one@one">
6.                <source  name="laboratory"/>
7.                <cible  name="laboratory"/>
8.                <transformation  operation="parity"/>
9.                <child_mappings map_id="m1.1"/>
10.               <child_mappings map_id="m1.2"/>
11.               <child_mappings map_id="m1.3"/>
12.       </mapping_element >
13.       <mapping_element  map_id="m1.1"  map_typ="one@one">
14.               <source  name="laboratory/name"/>
15.               <cible  name="name"/>
16.               <transformation  operation="parity"/>
17.               <child_mappings />
18.       </mapping_element >
19.       <mapping_element  map_id="m1.2"  map_typ="one@many">
20.               …
21.       </mapping_element >
22…
23.</mapping_structure >
```

FIG 4 – A structured mapping example.

5 Generating XSLT transformation

5.1 The XSLT generator

An XSLT generator is designed to generate XSLT stylesheets from the structured mapping established in section 4. For each matching node pair, the XSLT generator traverses the both the target schema graph and the mapping result in a depth first manner and generates template rules. The following describes the general algorithm used by the XSLT generator in order to translate a mapping result specification into XSLT templates. Our algorithm takes as input the target schema graph, source and target instances, and the mapping result specification and produces an XSLT stylesheet consisting of a series of templates rules, implementing the transformation. The XSLT generator proceeds in two steps:

Step 1: initialize the generation

a. The XSLT generator tries to locate the first node of the target graph having a match candidate.

b. For non mapped nodes, it acts as follow: if non mapped target node is not mandatory, or optional, nothing is generated, otherwise just element tags are generated in order to ensure the validity (against the target schema) of the produced instance. Once the first mapped node is localized, a ***nodes to process queue*** is initialized and the template rules generation can begin.

Step 2: Traverse the target schema graph and the mapping result

c. Generate a construction template for current node by using current mapping element.

d. For each ***child_mappings*** of the current mapping element, adjust the above template by inserting more construction or ***apply-template*** rules whenever necessary.

e. For the case of atomic node, insert new construction rule and no further process is needed for this node.

f. Add adjusted templates into XSLT stylesheet.

g. If a non mapped node is encountered, act as is ***step 1.b***.

h. If ***node to process queue*** is non-empty, extract one node as current node and loop back to ***step 2.a***, else return the generated XSLT stylesheet.

5.2 Example of XSLT generation

Let us consider the same source and target schema graphs of figure 3 and the mapping result of figure 4. The two root elements $laboratory_S$ and $laboratory_T$ match each other, thus a match template will be defined for driving instances of $laboratory_T$ from instances of $laboratory_S$. XSLT generator traverses the target schema graph, when node $laboratory_T$ is visited ***line 4*** is generated specifying the template will be instantiated when nodes satisfying the pattern (i.e. an XPath expression) ***laboratory*** are encountered. Meanwhile ***line 5*** and ***9*** are generated specifying that these markups will appear literally in output document. Next, since the current mapping element ***laboratory*** is a complex node. The XSLT generator prepares then for further construction of the children nodes by generating ***apply-templates*** instructions using the XPath expression in the corresponding mapping elements (lines 6 to 8).

```
1. < ?xml version="1.0" encoding="UTF-8" ?>
2. <xsl:stylesheet version="1.0" xmlns:xsl=http://www.w3.org/1999/XSL/Transform>
3. <xsl:output method="xml" version="1.0" encoding="UTF-8" ident="yes"/>
4. <xsl:template match="laboratory">
5.     <laboratory xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
                        xsi:noNamespaceSchemaLocation="laboratory_target.xsd">
```

```
11. <xsl:template match="name">
12.         <xsl:element name="name">
13.                 <xsl:value-of select="."/>
14.         </xsl:element>
15. </xsl:template>
```

Next, the node *address*$_T$ is processed. Since *address* matches split values *(street, city and zip)* of *location*$_S$, the needed enclosing element tags are generated and XSLT instructions that mimic the splitting of a string are introduced (lines 17 to 21).

```
16. <xsl:template match="location">
17.         <address>
18.                 <street> <xsl:value-of select="substring-before(substring-after(., ' '), ' ')"/> </steet>
19.                 <city> <xsl:value-of select="substring-after(substring-after(., ' '), ' ')"/> </city>
20.                  <zip> <xsl:value-of select="substring-before(., ' ')"/> </zip>
21.         </address>
22. </xsl:template>
```

The algorithm iteratively generates more templates to achieve the remaining constructions.

6 Conclusion

We presented a framework for structuring the mapping result and automatically generate XSLT programs. The proposed mapping result consists on several mapping elements, each which covers four dimensions: *entity* dimension specifying the source and target schema entities involved mapping element, the *cardinality* dimension specifying the nature of the mapping element (direct or complex), the *transformation* dimension specifying the used transformations operations, and the *structural* dimension specifying the elementary mapping elements may be combined into more complex mapping elements. Based on such structure, we show that we are able to generate automatically an XSLT program that translates a source instance into a target one.

References

1. Bergamaschi, S., Castano, S., Vimeracati, D.C.D., and Vincini, M.: An intelligent approach to information integration. Int. Proc. Conference on Formal Ontology in Information Systems, (1998) 253–267.
2. Castano, S., and Antonellis, V.D.: A schema analysis and reconciliation tool environment for heterogeneous databases. Int. Proc. Database Engineering and Applications Symposium, (1999) 53–62.
3. Doan, A., Madhavan, J., Domingos, P., and Halevey, A.: Reconciling schemas of disparate data sources: A machine Learning Approach. Int. Proc. ACM SIGMOD conference, (2001) 509–520.
4. Madhavan, J., Bernstein, P., and Rahm, E.: Generic schema matching with cupid. Int. Proc. VLDB'01, (2001).
5. Maedche, A., Motik, B., Silva, N., and Volz, R.: MARFA: A Mapping FRAmwork for distributed ontologies. Inter Proc. EKAW'2002, (2002) 235–250.
6. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity Flooding: A versatile Graph Matching and its Application to Schema Matching. Data Engineering, (2002).
7. Popa, L., Velegrakis, Y., Miller, R.J., Hernandez, M.A., and Fagin, R.: Translating Web Data. Int. Proc. VLDB'02, (2002) 598–609.
8. Rahm, E., and Bernstein, P.: A survey of approaches to automatic schema matching. In VLDB Journal, (2001) 334–350.
9. Rodriguez-Gianolli P., and Mylopoulos, J.: A Semantic Approach to XML-based Data Integration. Int. Proc. ER'01, (2001) 117–132.
10. Xu, L., and Embley D.W: Discovering Direct and Indirect Matches for Schema Elements. Int. Proc. DASFAA 2003, (2003) 39– 46.
11. XSLT 1.0. W3C Recommendation. XSL Transformations XSLT Version 1.0, Available at http://www.w3.org/TR/xslt, (1999).
12. XML Schema. (2001) W3C Recommendation, "XML Schema Primer", W3 Consortium, available at http://www.w3.org/TR/xmlschema-0, (2001).
13. Zerdazi, A. and Lamolle, M. : Modélisation des schémas XML par adjonction de métaconnaissances sémantiques. In ASTI'05, (2005) 29–32.
14. Zerdazi, A. and Lamolle, M.: Matching of Enhanced XML Schema with a measure of structural-context similarity. Inter Proc. WEBIST'07, (2007).