Schema Matching for Transforming Structured Documents

Aida Boukottaya
HEC/UNIL (University of Lausanne) &
Global Computing Center
EPFL (Swiss Federal Institute of Technology)
1015 Lausanne, Switzerland
Aida.Boukottaya@epfl.ch

Christine Vanoirbeek
Media Research Group &
Global Computing Center
EPFL (Swiss Federal Institute of Technology)
1015 Lausanne, Switzerland
Christine.Vanoirbeek@epfl.ch

ABSTRACT
Structured document content reuse is the problem of restructuring and translating data structured under a source schema into an instance of a target schema. A notion closely tied with structured document reuse is that of structure transformations. Schema matching is a critical strep in structured document transformations. Manual matching is expensive and error-prone. It is therefore important to develop techniques to automate the matching process and thus the transformation process. In this paper, we contributed in both understanding the matching problem in the context of structured document transformations and developing matching methods those output serves as the basis for the automatic generation of transformation scripts.

Categories and Subject Descriptors
I.7.2 [**Document and Text Processing**]: Document Preparation.

General Terms
Documentation, Languages.

Keywords
Document Structure Transformations, Schema matching.

1. Introduction
The need for developing methods and tools, that support data exchange and reuse has increased over the years, especially with the proliferation of Web data sources deploying a variety of data models and encoding syntaxes. XML (eXtended Markup Language) has clearly emerged as the most relevant standardization effort for structuring document and data on the Web; it leverages a promising consensus on the encoding syntax for both human and machine. However, reusing XML documents remains a challenging task. In XML document content reuse, a document (or a part of document) structured under one schema must be restructured and translated into an instance of a different schema. Thus, a notion tied to structured document reuse problem is that of structure transformations. This is typically attained in real world by writing translators encoded on a case-by-case basis using specific transformation languages. Currently the best known and widely adopted language for transforming structured documents is XSLT [22]. However XSLT is a powerful transformation language, it has several drawbacks. Simple transformations require the user to write a program, which needs non-trivial programming skills. Faced with the complexity of current structure transformation languages, several simpler and highly declarative transformation languages have been introduced. These languages try to keep a manageable balance between complexity and expressiveness. Authors in [19] propose a new language called *Paired SynTrees* which extends TT grammar with XPath expressions and a set of boolean conditions (including existence testing expressions and function constraints) in order to localize nodes in a tree. Special graphical tools have been also proposed to assist the specification of the transformations [15], [25]. Authors in [20] give an overview of existent structure transformation languages and tools.However, such languages and tools shield the user from programming effort; they require that a mapping between each source and target XML representations is carefully specified. Manual mapping is time consuming and thus especially unacceptable for applications where the information sources change frequently. Moreover, since the XML schemas can be very diverse, the mappings created by the expert are often complex. This complexity makes them hard to maintain when original XML schemas change.

This paper seeks to automating the process of mapping discovery (identified as *schema matching process*) and thus automatically deducing from such mappings the transform scripts which can rearrange and modify the associated data. The paper is organized as follow: section 2 describes related work. In section 3 and 4, we respectively propose a data model for XML Schemas and point a set of structure transformation operations. Section 5 details the proposed matching techniques. An evaluation study is presented in section 6. Finally, section 7 describes briefly the mapping structure and XSLT generation.

2. Related Work
Several approaches focusing on automating XML document transformations has been recently proposed by the document community. Examples include the work done in [11], where authors propose a syntax directed approach for automating structure transformations between two grammars based on finite state tree transducer. The idea behind this work is to generate a transformation semi-automatically if the user defines a matching between elements containing the text of the document (i.e. leaves). This approach presents several limitations: first it works only if the two grammars have common parts, which restricts the scope of transformations to local transformations. Moreover, this approach is unable to resolve all the heterogeneities that may

DocEng'05, November 2–4, 2005, Bristol, United Kingdom.

occur between structured documents: authors restrict themselves to transformations for which certain types of structure elements in the source document are transformed to the same types of elements in the target document. For example, a list of repeating nodes in the source document can only be transformed into a list that contains the same number of repeating elements. The authors of [18] propose an approach for automating the transformations of XML documents. To this end they define a set of DTD transformation operations that establish the semantic relationships between two DTDs. The approach is based on a tree matching algorithm, called *DMatch* (DTD Match), to discover automatically a sequence of operations that transforms a source DTD into a target DTD. The matching process is based on both provided auxiliary semantic information and a cost model. The latter model is based on heuristics functions for choosing transformation operations among multiple alternatives. This approach presents several limitations. First the matching algorithm used is only able to discover one-to-one correspondences between DTDs and does not deal with many-to-many matches. Second, the matching algorithm requires additional semantic information to work correctly, which limits the scope of its application since such semantic information is not always available. Finally, the matching algorithm used is inspired by work done in tree matching and is unable to deal with the current XML schema model.

Additionally to document community, database and artificial intelligence communities have widely considered schema matching problem in many application domains such as data integration, and peer-to-peer data management [5], [6], [9], [16], [17] . With the growing use of XML, several matching algorithms take into consideration the hierarchical structure of XML. In the following, we present some examples and check their applicability in the context of XML data transformations.

Cupid (Microsoft Research)

Cupid is a hybrid matcher combining several matching methods [12]. Cupid transforms the original XML schemas into trees and then performs a bottom-up structure matching. The basic assumption behind the structure matching phase of Cupid is that much of the information content is represented in leaves and that leaves have less variation between schemas then internal structures. Thus the similarity of inter-nodes is based on the similarity of their leaf sets. Schema structure in Cupid is used as a matching constraint, that is, the more the structures of the two nodes are similar, and more the two nodes are similar. For this reason, Cupid faces problems in the cases of equivalent concepts occurring in completely different structures, and completely independent concepts that belong to isomorphic structures.

Similarity Flooding (Stanford Univ. and Univ. of Leipzig)

In [13], authors present a structure matching algorithm called Similarity Flooding (SF). For computing structural similarities, SF relies on the intuition that nodes of two distinct graphs are similar when their adjacent nodes are similar. The spreading of similarities in the matched models is reminiscent to the way how IP packets flood the network in broadcast communication. An iterative process is used to propagate similarities between nodes, where in every iteration the similarity of a map pair is incremented by the similarity of its neighbours. An important assumption behind the algorithm is that adjacency contributes to similarity propagation. Thus, the algorithm will perform unexpectedly in cases when adjacency information is not preserved. Furthermore, SF ignores all type of constraints while performing structural matching. Constraints like typing and integrity constraints are used at the end of the process to filter mapping pairs with the help of user.

As we can see, proposed structural matching methods remain insufficient and very limited (generally deals only with DTDs and exploits few structural characteristics, essentially parent-child relationships). Convinced that the structural organisation in XML documents inferred some semantics of the data and traduced the designer point of view, a solution to XML schema matching problem should exploit this information in a manner that increase the matching accuracy. Furthermore, current schema matching algorithms only focus on discovering 1-1 mappings, also called *direct mapping*. The output result is a confidence score (ranging in [0,1]) between schemas' elements. When addressed the problem of automating document transformations, such output is often insufficient. First, because *complex mappings* (involving more than one source and/or target elements) make up a significant portion of discovered mappings in practice. Second, to generate a transformation script, scores between 0 and 1 are insufficient (we need to further precise transformation operations). In the remaining, we propose different methods and algorithms to solve such problems.

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 [23] 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 [2]. We model XML schemas as a directed labelled graph with constraint sets; so-called *schema graph*. Figure 1 illustrates a schema graph example.

3.1 Schema graph nodes

We categorize nodes into *atomic nodes* and *complex nodes*. Atomic nodes have no edges emanating from them. They are the leaf nodes in the schema graph. Complex nodes are the internal nodes in the schema graph. Each atomic node has a *simple content*, which is either an *atomic value* from the domain of basic data types (e.g., string, integer, date, etc.); or a *constructional value*, meaning a *list value* or a *union value*. The content of a complex node, called *complex content*, refers to some other nodes through directed labelled edges. In Figure 1, nodes *University* and *Library* are complex nodes, while nodes *Name* and *Location* are atomic nodes.

3.2 Schema graph edges

Each edge in the schema graph links two nodes capturing the structural aspects of XML schemas. We distinguish three kinds of edges: (1) *containment relationship,* denoted *c*, that is a composite relationship, in which a composite node ("whole") consists of some component nodes ("parts"); (2) *of-property* relationship, denoted *p*, that specifies the subsidiary attribute of a node; and (3) *association relationship*, denoted *a*, that is a structural relationship, specifying that both nodes are conceptually at the same level. Association relationships essentially model

key/keyref and substitution group mechanisms. They are generally bidirectional. Two association relationships are represented in Figure 1. The first association between the two nodes *Book* and *Monograph* is used for modelling the substitution group relation between the two nodes. While the second association relation between *Journal-article* and *Journal* specifies a key/keyref relation. Visually association edges are depicted as dashed lines.

3.3 Schema graph constraints

Different constraints can be specified with XML Schema language. These constraints can be defined over both nodes and edges. Typical constraints over an edge are cardinality constraints. Cardinality constraints over a containment edge specify the cardinality of a child with respect to its parent. Cardinality constraints over an of-property edge imply for example an optional or mandatory attribute for a given node. The default cardinality specification is [1..1]. We also distinguish three kinds of constraints over a set of edges: (1) *ordered Composition,* defined for a set of containment relationships and used for modelling XML Schema "sequences" and "all" mechanisms; (2) *exclusive Disjunction,* used for modelling the XML Schema "choice" and applied to containment edges; and (3) *referential constraint,* used to model XML schema referential constraints. Referential constraints are applied to association edges, and are generally modelled through a join predicate. As example, the association edge between A*rticle* and *Journal* with the predicate *Article/JournalRef = Journal* (as typical join condition). Other constraints are furthermore defined over nodes. Examples include *uniqueness* and *domain constraints.* Domain constraints are very broad. They essentially concern the content of atomic nodes. They can restrict the legal range of numerical values by giving the maximal/minimal values; limit the length of string values, or constrain the patterns of string values.

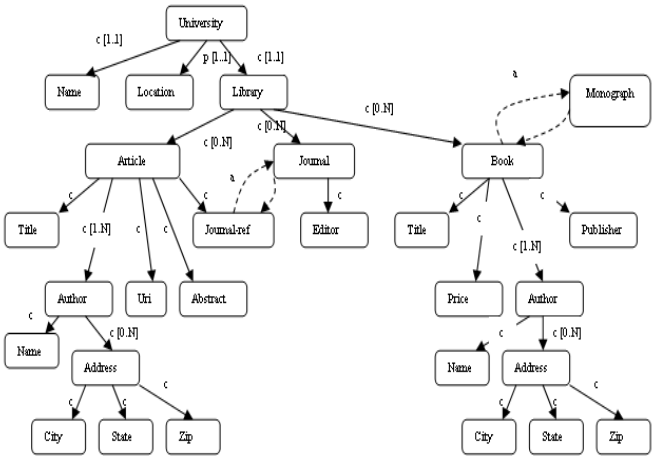


Figure 1. A Schema Graph example.

3.4 Type Binding and Type Hierarchy

In addition to elements and attributes declarations, XML schema introduced typing mechanisms. Built-in simple types as well as their restrictions are represented as domain constraints within the schema graph. User-defined types are not represented in the schema graph but represented in a type table and a binding from schema graph nodes to types are established. The concept of Schema graph as described above does not also include features like type extensions and abstract types. Extending a type means to add elements and/or attributes, which always results in a complex type. At a conceptual level, this refers to a generalisation/specification relationship. Such relationships are represented by the mean of a type hierarchy. Let us assume for example that we have an abstract type PUBLICATION and two subtypes of PUBLICATION respectively ARTICLE and BOOK. This will be represented in a type hierarchy by two specification relationships respectively between PUBLICATION and ARTICLE and PUBLICATION and BOOK. We also specify that PUBLICATION type is abstract (meaning that it may not have direct instances, but its concrete subtypes may). If nodes *Article* and *Book* in the schema graph of Figure 1 are of types ARTICLE and BOOK, a binding between types and schema graph nodes is generated.

4. Transformation operations

To motivate our choice of transformation operations, we list the following problems we must face in matching two XML schemas:

- Frequently, schema designer tends to qualify semantically similar concepts using different names. For this we consider a rename operation defined as follow: **Rename:** *t = rename (s)*, generates a construction[1] that is the same as a construction s, but with a different name t.
- For a target node, a particular source may have (1) a proper *subset* of the desired values or (2) a proper *superset* of the desired values. For such case, we propose two operations *Union* defined as follow: **Union:** $t = union\ (s_1,s_2)$, generates a construction *t* whose content is the union of s_1 values and s_2 values; and *Selection* defined as follow: **Selection:** $t = Select_P\ (s)$ (where *P* is a predicate) generates a construction *t* whose content is the part of content of *s* that satisfies the predicate *P*.
- Schema designers do not always choose to represent values at the same level of atomicity. For example, an author name is represented in a given schema using an element *Name*. While in an other schema, it is separated into a *First-Name* and a *Last-name*. For such cases, we define two operations Merge and Split as follow: **Merge:** $t = merge\ (S_1....S_i)$, generates a construction *t* whose value is obtained by concatenating $s_1....s_i$ values; and **Split:** $(t_1...t_i) = split_{criteria}\ (s)$, where $t_1...t_i$ are obtained by splitting a construction *s* respecting to a separation criterion. An example of separation criterion is "white space" in the case of strings.
- A target construction may be obtained by applying a natural join (as in relational schemas) to source constructions. We define a join operation as follows: **Join:** $t= join_P(S_1,S_2)$, generates a target construction *t* which is the natural join of S_1 and S_2 under the predicate *P*.
- Frequently we need some specific functions to transform the content of source values into target values. Such functions include for example unit conversion, date format

[1] Construction refers to schema elements, or attributes, or relationships.

transformations, mathematical functions such as min, avg, div, etc. For such cases, we introduce a new operation apply defined as follow: **Apply:** $t = apply_f\ (s_1 \ldots s_i)$ where f is a function that takes $s_1 \ldots s_i$ values and returns t, whose value corresponds to $f(s_1 \ldots s_i)$.

- For the case where we have a one-to-one matching without any modification, we provide an operation called connect: **Connect:** $t = connect\ (s)$, generates a construction t which has the same content and label as s.

5. Matching Process

To match schema graphs, we make use of four basic matching criteria (1) linguistic matching, (2) datatype compatibility, (3) Designer type hierarchy and (4) structural matching.

5.1 Linguistic matching

The aim of this phase is to compute the similarity between schema nodes based on the similarity of their labels. To perform linguistic matching, we make explicit the meaning of used element names and establish semantic relationships between them based on WordNet [8]. Most of schema matching algorithms suggest the use of WordNet for linguistic matching, but generally gave few, if any details about how they exploit WordNet. Our linguistic matching is inspired essentially from Hirst and St-Onge's work [14]. When attempting to find a relation between two words, each synset (set of synonyms representing a sense associated to a word) of the first word must be considered with each synset of the second word, looking for a possible connection between some synset of the first word and some synset of the second. The idea behind Hirst and St-Onge's measure of semantic relatedness is that two concepts are semantically close if their WordNet synsets are connected by a path that is not too long and that does not change direction too often. A set of allowable paths have been then defined. The linguistic similarity between two words is computed based on the path relating them as follows:

$$c - Path\ length - k \times number\ of\ changes\ of\ direction^2$$

Moreover, based on the classification of allowable paths, we identified four kinds of semantic relations between words, namely *equivalent* (≡), *Broarder than* (⊇), *Narrower than* (⊆), and related *to* (~). The detailed algorithm is given in [2]. The same algorithm is also applied to type names. To simplify the comprehension of our approach, we assume in this paper that nodes have the same names as their types.

5.2 Datatype compatibility

XML schema recommendation provides many different datatypes and regular expressions. It is probably the ideal set of datatypes since it is refined enough for the purpose of schema matching. In fact, XML schema allows the definition of very specific datatypes. For this, we make use of built-in XML schema datatypes hierarchy [24] in order to compute datatype compatibility coefficient. XML schema datatypes are classified in multiple categories (called primitive datatypes) including for example Duration, Boolean, String, Decimal, etc. Each category has several derived datatypes. Two datatypes are considered to be similar if they belong to the same datatype category, and their datatype compatibility depends on their respective position in XML schema datatype hierarchy. Based on XML Schema datatype hierarchy, we construct a datatype compatibility table, such as the one used in [12] that gives a similarity coefficient between two given datatypes. Moreover, we also make use of imposed constraints (expressed by means of facets) over datatypes in order to refine the datatype compatibility coefficient (two datatypes belonging to the same category and presenting similar set of constraints are more likely to be similar)[3]. For example, two string datatypes (or string derived datatypes) having similar length constraints and two integers having similar numerical value ranges are more likely representing similar real word entities. We limit the scope of datatype compatibility to atomic nodes that are already similar using linguistic matching method. Finally we update the linguistic similarity coefficient of atomic nodes by including their datatype compatibility.

5.3 Designer Type hierarchy

As mentioned in section 3.4, XML schema features concerning sub-typing, abstract types and substitution group mechanisms traduce the designer point of view and could be used as a set of meta-data to help the matching process to discover both direct and complex mappings. In the following, we present some examples of such features and show how they can be used to deduce match candidates: (1) Abstract Types: Let us consider a source schema where two elements *Journal-Article* and *Procceding-Article* are declared respectively of types JOURNAL-ARTICLE and PROCEEDING-ARTICLE (we use the same appellation for elements and types to simplify the comprehension of the example). Assume that these two types are subtypes of an abstract type ARTICLE. Consider a target schema where only an element *Article* of type ARTICLE is presented. Based on the fact that JOURNAL-ARTICLE and PROCEEDING-ARTICLE are subsets of type ARTICLE in the source schema and the type ARTICLE in the source schema matches the type ARTICLE in the target schema, one can deduce the following complex mapping: the *union* of source elements *Journal-article* and *Proceeding-article* matches the target element *Article*. This kind of hints may also provide wrong matches, let us keep the same source schema, but consider the target schema as the schema represented in Figure 1, element *Article* in the target schema corresponds to element *Journal-Article* in the source schema and not to the union of elements *Journal-article* and *Proceeding-article*. Such wrong matches can be corrected by the structural matching techniques described in section 5.4.

[2] c and k are constants. The choice of c and k were done on running experiments based on examples provided in [3] where authors compare several semantic distance measure algorithms. C=8 and k=1, thus, the longer the path and the more changes of direction, the lower the weight.

[3] Although XML Schema offers specific datatypes, those are usually not exact and constraints are often incomplete, since they are not a necessity, but merely a convenience for the schema designer. Our use of datatypes constraints is restricted to some facets. For example, we do not consider patterns comparison. Works such as [7] and [26] can be used to extend datatype compatibility measure.

(2) Substitution Group: Two substitutable elements are conceptually at the same level. Let us consider a source schema where elements *book* and *monograph* are substitutable and a target schema where element *publication* is similar (by linguistic matching) to the source element *book*. Since in the source schema elements *book* and *monograph* are substitutable, a direct match between *monograph* and target schema element *publication* can be inferred.

The result of this step is a set of direct and complex mappings (essentially involving Union/Selection operators). Such mappings will be kept or rejected using either structural matching techniques or user intervention. In the case where type hierarchy is not available, we also make use of semantic relationships discovered by the linguistic matching to derive complex matches. However, we give the priority to designer type hierarchy since it reflects the designer point of view. More examples and algorithms on how to derive match candidates based on type hierarchy are detailed in our previous work [2].

5.4 Structural matching

The matching techniques described in sections 5.1, 5.2 and 5.3 may provide incorrect match candidates. Structural matching is used to correct such match candidates based on their structural context and thus derive correct direct and complex matches. Structural matching relies on the notion of *node context*. In the following we describe the basis needed to define such context and thus perform structural matching.

5.4.1 Node context definition

As in [10], we distinguish three kinds of node *contexts* depending on its position in the schema graph: (1) *The ancestor-context:* of a node *n* is defined as the path (going through containment edges) having *n* as its ending node and the root of the schema graph as its starting node. The ancestor-context of the root node is empty and it is assigned a NULL value; (2) *The Child-context*: of a node *n* includes its attributes (through of-property edges) and its immediate subelements (through containment edges). The child-context of a node reflects its basic structure and its local composition. The child-context of an atomic node is assigned a NULL value. For association relationships, we include the associated nodes in the child context. For example the child-context of the node *Article* in the schema graph of Figure 1 is composed of nodes *Title*, *Author*, *Uri*, *Abstract* and *Journal-ref*. Since *Journal-ref* is a key ref node, we also include the referential node *Journal* in the child-context of *Article*; and (3) *The leaf-context:* Leaves XML documents represent the atomic data that the document describes. The leaf-context of a node *n* includes the leaves of the subtrees (composed by containment relationships) rooted at *n* and. The leaf-context of an atomic node is assigned a NULL value.

The context of a node is defined as the union of its ancestor-context, its child-context and its leaf-context. Two nodes are structurally similar if they have similar contexts. To measure the structural similarity between two nodes, we compute respectively the similarity of their ancestor, child and leaf contexts.

5.4.2 Path resemblance measure

Structural node context defined in section 5.4.1 relies on the notion of *path*. In order to compare two contexts, we essentially need to compare two paths. Path comparison has been widely used in answering conjunctive queries. However, they rely on strong matching following the two classical constraints: root constraint and edge constraint. Under such conditions paths such as *Author/Publication* and *Publication/Author* are no matched however they convey same semantics. Other unmatchable paths under such conditions are *Author/Contact/Address* and *Author/Address*. Based on such observations, it is more appropriate to go beyond the strong matching by relaxing the above conditions. One can think of several ways of relaxing strong matching: for example allow matching paths even when nodes are not embedded in a same manner or in the same order. Several works in query answering have proposed relaxation issues to approximate answering of queries (including path queries) [1], [4]. Relaxations may give raise to multiple match candidates. For this reason, authors in [4] define a path resemblance measure between a given path query Q and a path in the source tree. Such measure is used for ranking match candidates. We extend these definitions by allowing two elements within each path to be matched, even if they are not identical but their linguistic similarity exceeds a fixed threshold. We define a *path resemblance measure*, denoted *pr*, which determines the similarity between two given paths. The values of *pr* range between 0 and 1. Match candidates can then be ranked according to *pr* measure.

Consider two paths P_1 and P_2 being matched (when P_1 is a target path and P_2 is a source path), P_2 is the best match candidate for P_1 if it fulfils the following criteria:

- The path P_2 includes most of the nodes of P_1 in the right order.
- The occurrences of the P_1 nodes are closer to the beginning of P_2 than to the tail, meaning that the optimal matching corresponds to the leftmost alignment.
- The occurrences of the P_1 nodes in P_2 are close to each other, which mean that the minimum of intermediate non matched nodes in P_2 are desired.
- If several match candidates that match exactly the same nodes in P_1 exist, P_2 is the shortest one.

To calculate $pr\ (P_1, P_2)$, we first represent each path as a set of string elements; each element represents a node name (e.g., the path *Author/Publication* is a string composed two string elements *Author* and *Publication*). We used the four scores established in [4] and borrowed from dynamic programming for string comparison; each of which corresponds to one of the above criteria.

5.4.2.1 Longest Common Subsequence

To answer the first criterion, we use a classical dynamic programming algorithm in order to compute the *Longest Common Subsequence* (LCS), between P_1 and P_2. More the length of the longest common subsequence is high; more P_2 includes P_1 nodes in the right order. Finally, to obtain a score in [0,1], we normalize the length of the longest common subsequence by the length of target path P_1 as following:

$$lcs_n(P_1, P_2) = |lcs(P_1, P_2)|/|P_1|$$

Example: Consider P_1 to be *Publication/Book/Author* and P_2 as *Author/Publication/Book*, the longest common subsequence between the two paths is *Publication/Book*, $|lcs\ (P_1, P_2)| = 2$, thus $lcs_n = 2/3 = 0.66$.

5.4.2.2 Average positioning

To answer the second criterion, we first compute, according to $lcs\ (P_1, P_2)$ what would be the average positioning of the optimal matching of P_1 within P_2. The optimal matching is the match that starts on the first element of P_1 and continues without gaps. Consider P_1 = Author/Publication/Book and P_2 = Staff/Author/Publication/Book, since the optimal matching corresponds to the leftmost alignment, the average optimal position, denoted *Av-Optimal-Position* is (1+2+3)/3 =2. We then evaluate using the LCS algorithm, the actual average positioning (*AP*). *AP* takes the value 3 in our example ((2+3+4)/3). Last, we compute *pos* coefficient indicating how far the actual positioning is from the optimal one, using the following formula:

$$pos(P_1, P_2) = 1-((AP-Av\text{-}Optimal\text{-}Position)/(|P_2|-2*Av\text{-}Optimal\text{-}Position+1))$$

5.4.2.3 LCS with minimum gaps

To answer the third criterion, we use another version of the LCS algorithm in order to capture the LCS alignment with minimum gaps. If P_1 = Person/Address and P_2 = Person/Contact/Address, we count a gap of length 1 between the two paths, thus *gaps* =1. To ensure that we obtain a score inferior to 1, we normalize the obtained gap using the following formula:

$$gap(P_1, P_2) = gaps/(gaps + lcs(P_1, P_2))$$

5.4.2.4 Length difference

Finally, in order to give higher values to source paths whose length is similar to the target path, we suggest to compute the length difference *ld* between a source path P_1 and $lcs(P_1, P_2)$ normalized by the length of P_1 as follow:

$$ld(P_1, P_2) = (|P_2| - lcs(P_1, P_2))/|P_2|$$

To obtain the path resemblance score, all the above metrics are combined as follow:

$$pr(P_1, P_2) = \alpha\ lcs_n\ (P_1, P_2) + \beta\ pos(P_1, P_2) - \lambda\ gap(P_1, P_2) - \delta\ ld(P_1, P_2)$$[4]

Example: Let P2 = Author/Book/title and P1= University/Author/Publications/Book/Description/Title/subtitle;
We have $|lcs| = (2+3+4)/3 = 3$, AP = (2+4+6)/3 = 4, gaps =2, ld= 7-3/7=4/7. We obtain a path resemblance score equal to 0.68.

5.4.3 Structural context similarity

5.4.3.1 Ancestor context similarity

The ancestor context similarity, *ancestor-ctx-sim* captures the similarity between two nodes based on their ancestor context; defined for a given node *n* by the path from the root to *n*. The *ancestor-ctx-sim* between two nodes n_1 and n_2 is given by the path resemblance measure between the two paths *(root, n_1)* and *(root, n_2)* weighted by the linguistic similarity between n_1 and n_2 as follow:

$$ancestor\text{-}ctx\text{-}sim\ (n_1, n_2) \leftarrow pr\ ((root, n_1), (root, n_2)) \times lsim\ (n_1, n_2)$$

5.4.3.2 Child-context similarity

The child-context similarity, *child-ctx-sim* is obtained by comparing nodes immediate descendents (children) sets including attributes and subelements. Let us consider a node n_1 having n immediate children represented by the set (n_{11}, …, n_{1n}) and node n_2 having m immediate children represented by (n_{21}, …, n_{2m}). To compute the similarity between these two sets, we first compute the linguistic similarity between each pair of children in the two sets. Second, we select the matching pairs with maximum similarity values. And finally, we take the average of best similarity values.

5.4.3.3 Leaf context similarity

Since the effective content of a node is often captured by the leaf nodes of the subtree rooted at that node, we compute leaf context similarity of two nodes n_1 and n_2 by comparing their respective leaves sets: leaves (n_1) and leaves (n_2). It is possible that each schema represents different levels of abstraction and different granularities. Thus, to compute the similarity between two leaves $l_1 \in$ leaves (n_1) and $l_2 \in$ leaves (n_2), we propose to compare the contexts in which appear these leaves. If a leaf node $l \in$ leaves (n_1), then the context of l is given by the path from n_1 to l. The context similarity of two leaves is then obtained by comparing such paths; the path resemblance measure is then used as follow:

$$Leaf\text{-}sim\ (l_1,l_2) = pr\ ((n_1,l_1), (n_2,l_2)) \times lsim\ (l_1,l_2)$$

The leaf context similarity of two nodes n_1 and n_2 is obtained by first computing the leaf similarity between each pair of leaves in the two leaves sets, second selecting the matching pairs with maximum similarity values, and finally taking the average of best similarity values.

5.4.4 Node similarity

In this section, we propose to compute the similarity of two nodes belonging respectively to source schema graph and target schema graph by combining all the previous similarity measures (linguistic similarity, datatype compatibility and context similarity). For this, we distinguish three different cases:

- *Case 1: The two nodes being compared are atomic nodes* (leaves), and then their respective child context and leaf context are assigned the NULL value. The similarity between two atomic nodes is then given by the similarity of their respective ancestor context weighted by their linguistic similarity.
- *Case 2: One of the two nodes being compared is an atomic node, say n_1, and the other is a complex node, say n_2.* Since for the atomic node n_1, the child-context and the leaf-context are assigned a Null values. The similarity between n_1 and n_2 is obtained by computing first their respective ancestor-context. Second, since the content of an atomic node is captured by the node it self, while the content of a non

[4] α ,ß, λ and δ are positive parameters ranging between 0 and 1 that represent the comparative importance of each factor. They can be tuned but must satisfy a + ß = 1, so that pr(P1, P2) =1 in case of a perfect match, and λ and δ must be chosen small enough so that pr cannot take a negative value.

atomic node is captured by its leaf-context, we propose to calculate the average of the linguistic similarity between n_1 and nodes belonging to the leaf-context of n_2. The similarity between the two nodes is then obtained by weighted similarity of their ancestor and leaf contexts.

- *Case 3: Both nodes are complex nodes* and then their similarity is the weighted sum of their ancestor context similarity, their child-context similarity and their leaf context similarity.

Once element similarity is computed, one can use it to correct indirect matches and set appropriate operations as we will describe in section 5.4.5.

5.4.5 Discovery of nodes and edges matches

Most schema matching algorithms produce similarity scores between source and target schemas nodes such as the ones we produce in section 5.4.4. Such result solves partially the problem. First, produced similarities between individual nodes are not enough to produce access paths for retrieving data from the available sources. Second, all the produced mappings are one-to-one mappings, complex mappings identified using type hierarchy have to be incorporated in the matching result and further complex mappings have to be discovered. For this we proceed in four steps:

Step 1: Compatible nodes identification

While generating mapping elements, we apply a *top-down strategy*[5]. At the top level, we establish correspondences between complex nodes of the target and source schemas. Similar complex nodes are called *compatible nodes*. Let us consider simplified schema graphs illustrated by Figure 2(a) and 2(b). Assume that both nodes *University* are similar (based on node similarity measure described in section 5.4.4), then they are considered as *compatible nodes*. Node *Library* is not a compatible node, since it has not similar node. Visually compatible nodes are depicted as coloured boxes and dashed lines.

Step 2: Context generation for compatible nodes

After identifying compatible nodes, we proceed to construct a context for each compatible node (the notion of context here differs from the context we defined in section 5.4.1). By taking edges around a complex node *n* into account, we cluster a set of nodes and edges with a complex node as a conceptual component in the schema graph. We call this the context of *n*. For a given compatible node *n*, we construct such context by (1) including all atomic nodes directly related to the compatible node *n*; (2) including all non compatible nodes directly connected to *n* with their connected atomic nodes and connected non compatible nodes; (3) if a directly connected compatible node is also similar to an atomic node, it is also included in the context of *n*; (4) including all nodes having an association relationship with *n* and their respective context; and (5) including all containment relationships between nodes in the context of *n*. Figure 2 illustrate two schema graphs after context construction.

Example: In Figure 2 (b), the context of the compatible node *University* includes atomic nodes *Name* and *Location* and non compatible node *Library*. The context of compatible node *Article* includes referential node *Journal* and its context. In Figure 2 (a), the context of node *University* include the compatible node *Address*, since *Address* is similar to a leaf node (*location*) belonging to the context of a matched node *University*.

Step 3: Node mappings generation

At this point, we finished with the top level comparison between source and target schema graphs. We are now ready to detect node and edges matches at the bottom level. For each matching pair (n_T , n_S) which represented two compatible nodes in source and target schema graphs, we make use of node similarity score generated in section 5.4.4 to settle nodes matches. The following gives examples on how we proceed:

Example 1: Let the schema in Figure 2 (a) be the target schema and the schema in Figure 2 (b) be the source schema. Consider the two compatible nodes: target node *University* (*University*$_T$) and the source node *University* (*University*$_S$), we first settle node-set matches between both source and target contexts that hold with the highest node similarity score. As an example, we settle the match pair ($Name_T$, $Name_S$) using a *connect* operation.

Example 2: The target node *Address*$_T$ is both similar to the source nodes University/Location and Author/Address with approximatively same scoring. This is due to in the case of *University/Location* to the fact that ancestor context similarity is high and in the case of *Author/Address* to the fact that the leaf context similarity is high. Since target node *University/Address* and source node *Author/Address* belong to non compatible complex nodes while target element *University/Address* and source element *University/Location* belong to two compatible contexts, a match is then derived between source node *University/Location* and target node *University/Address*. Moreover since we decide to map a non-leaf node with a leaf node, a complex mapping with *split* operation can be deduced.

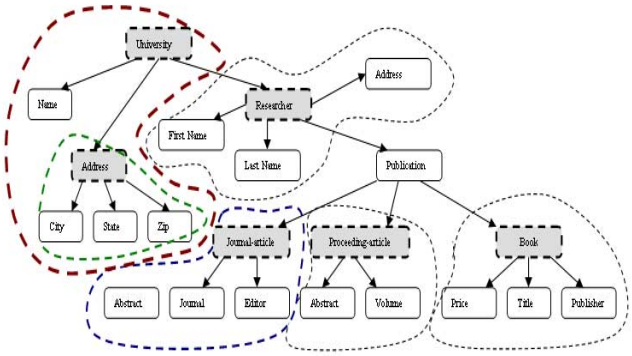
Example 3: Assume that we have already discovered that the union of target nodes *Journal-article* and *Proceeding-article* match the source node *Article* based on designer type hierarchy analysis. Such mapping can be confirmed or rejected by the system after compatible nodes context analysis. In fact, the context of source node *Article* includes the referential node *Journal*. Moreover, based on node similarity, the target node *Journal-article* is compatible with both source nodes *Article* and *Journal*. By analysing the contexts of source node *Article*, we discover that it more likely matches the target node *Journal-article*. The complex mapping is then removed and a new mapping is settled between source node *Article* and target node *Journal-article* using a *join* operation. Let just notice that if node *Journal* is not present in the source schema graph, the discovered complex mapping is accepted and a *selection* operation is assigned to it.

Step 4: Access paths generation

With the available correspondences between nodes in source and target schemas, we further discover matches between edges. As in

[5] We use the same top-down strategy as in [26]. However the difference is that this technique is used to discover structural similarity. In our approach, it is just used for mapping generation; the structural matching has been already performed.

[26], the recognition of edges matches starts by locating an edge set e_t in T. Then, based on nodes N_t connected by e_t, we can locate a set of nodes that correspond to N_t in S, from which we either locate or derive a edge set e_s that corresponds to e_t. We further focus on the discovery of access paths in order to retrieve source data when performing transformation. For each target element t, we first define the access path indicating where matched source elements are localized, then the discovered transformation operation and finally the conditions under which the mapping element holds true. Examples of generated mapping rules are given in Figure 3.



(a)

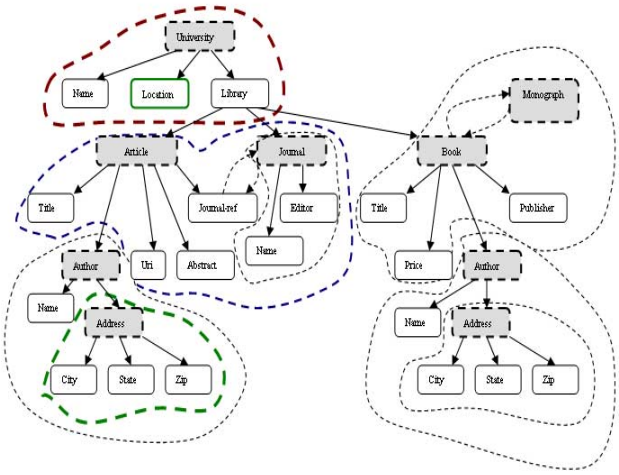


Figure 2. Source and target schema graphs after context construction.

Target	Source access paths	Transformation operation	Conditions
University	University	connect	-
Name	University/Name	connect	-
Address	University/Location	Split $_{ws}$ (Location)	-
City State Zip	University/Location University/Location University/Location	Split $_{ws}$ (Location) [1] Split $_{ws}$ (Location) [2] Split $_{ws}$ (Location) [3]	-
Researcher	University/Library/Article/Author University/Library/Book/Author University/Library/Monograph/Author	Union	-
First-name Last-name	Author/name Author/name	Split $_{ws}$ (Name) [1] Split $_{ws}$ (Name) [2]	-
Address	Author/Address/city Author/Address/State Author/Address/Zip	Merge	-
Publication	-	-	-
Journal-Article	University/Library/Article University/Library/Journal	Join	University/Library/Article/Journal-ref = University/Library/Journal/name
Abstract	University/Library/Article/Abstract	Connect	-
Journal	University/Library/Journal/Name	Connect	-
Editor	University/Library/Journal/Editor	Connect	-
Proceeding-Article	-	-	-
Abstract	-	-	-
Volume	-	-	-
Book	University/Library/Book | University/Library/Monograph	Connect	-
Price	University/Library/Book / Price University/Library/Monograph/ Price	Connect Connect	- -
Title	University/Library/Book /Title University/Library/Monograph/ Title	Connect Connect	- -
Publisher	University/Library/Book /Publisher University/Library/Monograph/ Publisher	Connect Connect	- -
University/Researcher/ Publication/Journal-Article	University/Library/Article/Author	-	Author = Researcher
University/Researcher/ Publication/Book	University/Library/Book/Author	-	Author = Researcher

Figure 3. Examples of generated mapping rules.

6. Performance Study

6.1 Real world example

In order to evaluate the proposed matching techniques, we considered one real-world application: *bibliographic data description*. The characteristics of used XML schemas are summarized in Figure 4 showing some indications of the complexity of test schemas. We choose schemas that differ in the number of nodes (schema size) and in their depth (the manner of nodes nesting). Test schemas present linguistic and structural heterogeneities. We let any one of the schema graphs be the target and let any other schema graph be the source. Different granularities and abstract levels are used to describe the same real world concepts. Test schemas require several indirect matches involving merge/split, union/select and join operators. To compute real matches, two different users were involved, and the average number of users discovered matches was considered. The total number of real matches was *1382* matches (*1102* direct matches and *280* complex matches). Our matching algorithm discovered 1312 matches including 1281 correct matches and 31 incorrect matches. The incorrectly classified mapping elements include 18 direct mappings and 13 complex mappings. The correctly recognized mapping elements included 1082 direct matches and 199 indirect matches. For direct matches, the precision, recall, F-measure and overall achieved 99%, 94%, 97%, and 93%. For complex matches, the precision, recall, F-measure and overall achieved 98%, 71%, 82%, and 70%. The performance of the matching algorithm reached for precision, recall, F-measure and overall 98%, 92%, 95%, and 90%.

Our process successfully found all the complex matches related to the problems of *Merged/Split Values* and *join relationships*. However, for the problem of *union/selection,* our matching algorithm correctly found all the complex matches related to 80 of 93 *union/selection* matches and incorrectly declared 13 extra *union/selection* operators. For discovering *union/selection* operators, we essentially rely on type hierarchy analysis, if available; otherwise we make use of WordNet semantic relationships. Among the 80 discovered *union/selection* relations, 22 are discovered using WordNet. The experimental results show

that the combination of linguistic and structural matching produces fairly reasonable results, even if schemas are structurally

Domain	Bibliographic Data							
Schemas	#1	#2	#3	#4	#5	#6	#7	#8
# Nodes	31	40	54	39	28	38	19	22
# Paths	30	39	53	38	26	37	18	20
Max Depth	4	8	10	6	6	8	4	3

Figure 4. Characteristics of tested schemas.

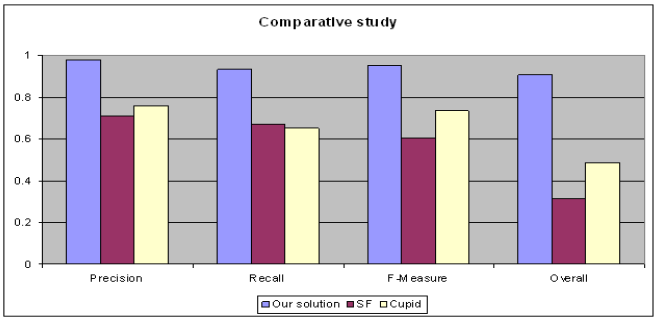


Figure 5. Comparative study with Cupid and SF.

highly heterogeneous.

6.2 Comparative study

In this section, we essentially run evaluation comparisons between our proposed solution, Cupid and Similarity Flooding systems. This is because Cupid, SF and our solution are fairly comparable because they deal with XML structure, they are all schema based, and they all utilize linguistic and structural matching techniques. Figure 5 illustrates the obtained results. From the point of view of the quality of the matching results, our proposed solution outperforms the other systems:

(1) ***Direct matches:*** Since our basic goal is to compare the structural matching capabilities of each system, we use the results of our linguistic matching algorithm as an initial mapping to both Cupid and SF. Given schemas of varying levels of details such as address (city, state, zip) and address, both Cupid and SF will return a low similarity measure. The reason is that Cupid is biased towards the similarity of leaf nodes, and SF towards the similarity of adjacent nodes. When matching schema elements with different contexts, such as researcher (name, address) and researcher (name, supervisor (name, address)), both Cupid and SF fail to distinguish researcher name from supervisor name. Our solution is able to obtain correct mappings because we maximize the use of structure by taking into considerations ancestor-context, child-context and leaf-context similarities.

(2) ***Complex matches:*** Up to now, most of current matching approaches have focused only on direct matches. They do not consider complex mapping. In Cupid, if a leaf node *s* in the source schema is highly similar to a target leaf node *t* in the target schema, a mapping element between *s* and *t* is returned. This resulting mapping may be 1:n, since a source element may map to many target elements. This simplest scheme to compute global 1:n mappings is very limited. First because only splitting values are considered (no n :1 mappings are discovered). Second, this technique leads frequently to wrong complex matches, imagine that we have a source schema with a node S.*University* having a child node *Address (S.Address)* and a target nodes T.*University* and T.*Author* where both of them having a child node *Address*. If the similarity between *S.University.Address* and *T.University.Address* approximates the similarity between *S.University.Address* and *T.Author.Address*, a wrong complex match will be discovered. This is avoided in our approach by considering only compatible nodes contexts. Finally, no union/select and join complex mappings are considered in Cupid. As in cupid, the discovery of complex match is SF is done after the structural matching. SF makes use of several filters to deduce the list of match candidates from a list of ranked matching pairs. The filtering can be characterized by providing a set of constraints and selection functions that pick the best subset of the multiple mapping under a given selection metric. For a given similarity threshold, SF selects a subset of a multiple mapping, in which all map pairs carry a similarity value of at least equal to the threshold value. Contrary to Cupid, SF can generate global m:n mappings, however similarly to Cupid several wrong mappings are generated and no specific operations are discovered. Our matching solution gives reasonably correct complex mappings because we limited the search scope to compatible nodes contexts and rely on structure to discover such mappings.

7. XSLT scripts generation

After validating generated mapping rules by the user, we structure the mapping result using W3C XML Schema language and this for two reasons. First, it is easier to manipulate structured mapping result either to modify it or to automatically generate transformation scripts. Second, structuring the mapping result greatly increases its reusability and adaptation, especially when schemas evolve. The nature of mapping result may be understood by considering different dimensions, each describing one particular aspect: (1) *the entity dimension*, specifying schema entities involved in a mapping element; (2) *the cardinality dimension*: determining the cardinality of a mapping element ranging from direct mapping (*1:1*) to complex mapping (*m:n*); (3) *the structural dimension*, reflecting the way how elementary mapping elements may be combined into more complex mapping elements; (4) *the transformation dimension*, reflecting how instances of the source schema are transformed during the mapping process; and (5) *the constraint dimension*, controlling the execution of a mapping element. Based on the established mapping between source and target schema in section 5.4, a mapping generator relates a given source and target schema graphs by generating an instance of the mapping schema containing a set of mapping elements each of which encapsulates all information needed to transform instances of source nodes into instances of target nodes. An XSLT generator will then traverse the both the target schema graph and the mapping result in a depth-first manner and generates adequate XSLT templates for each mapping element. The structure of the target schema is respected, while generating the XSLT script. This guarantees the generation of valid target instances after the transformation process. Structured mapping generator and XSLT generator descriptions are out of the scope of this paper. Detailed description and examples could be found in [2].

8. Conclusion

Due to the extensive use of XML markup language in several domains, there has been a great interest on proposing rich data models that reflects document semantics and structure. The existence of such rich models has made a large amount of heterogeneously XML documents widely available. In this framework, XML documents transformation is of major concern. Currently, to perform XML document transformations, the burden falls on the human to manual coding the transformations. This paper proposed novel schema matching techniques for automating the transformation of XML documents. We essentially proposed a structural similarity measure that relates schemas nodes based on the similarity of the structural context in which they appear. Our experiments showed that the combination of ancestor, child and leaf context play an important role in deriving correct matches. Generated mapping result is then used to automatically generate XSLT scripts. A prototype system that incorporates a conceptualization toolkit for generating and graphically represented schema graphs for W3C XML schema and graphical user interfaces to support the matching process and its validation, has been implemented.

9. REFERENCES

[1] S. Amer-Yahia, S.Cho, D. Srivastava, "Tree Pattern Relaxation" EDBT'02, 2002.

[2] A.Boukottaya. "Schema matching for structured document transformations", PHD thesis, October 2004.

[3] A. Budanitsky and G.Hirst. Semantic distance in WordNet. An experimental, application oriented evaluation of five measures, 2003.

[4] D.Carmel, N. Efraty, G.M.Landau, Y.S.Maarek, and Y.Mass. An Extension of the vector space model for querying XML documents via XML fragments. *Second Edition of the XML and IR Workshop, In SIGIR Forum*, Volume 36 Number 2, Fall 2002.

[5] S. Chawathe, H. García-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of heterogeneous information sources. In *16th meeting of the IPSJ*, pages 7-18, Tokyo, Japan, 1994.

[6] AH.Doan, P.Domingos, A.Halevey. Reconciling schemas of disparate data sources:A machine Learning Approach. In *Proceedings ACM SIGMOD Conference*, pages 509-520, 2001.

[7] D.W. Embley, D.M.Campbell, Y.S. Jiang, S.W.Liddle, D.W. Lonsdale, Y.K.Ng and R.D.Smith. Conceptual-model-based data extraction from multiple record Web pages. *Data and Knowledge Engineering*, 31(3), pages 227-251, 1999.

[8] Lexical chains as representations of context for the detection and correction of malapropisms. In: Christiane Fellbaum (editor), WordNet: An electronic lexical database, Cambridge, MA: The MIT Press, 1998.

[9] JA. Larson, SB. Navathe, R. ElMasri. A Theory of attribute equivalence in databases with application to schema integration. *IEEE TransSoftwareEng* 16(4):449-463, 1989.

[10] Mong Li Lee, Liang Huai Yang, Wynne Hsu, Xia Yang. XClust: Clustering XML Schemas for Effective Integration, in *11th ACM International Conference on Information and Knowledge Management (CIKM)*, McLean, Virginia, November 2002.

[11] P. Leinonen. Automating XML Document Structure Transformations. In *Proceedings of the ACM Symposium on Document Engineering*, France, 2003.

[12] J. Madhavan, P.A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2001.

[13] S.Melnik, H.Garcia-Molina, E.Rahm. Similarity Flooding: A versatile Graph Matching Algorithm and its Application to Schema Matching. In *Proceedings of the 18th International Conference on Data Engineering*, 2002.

[14] A.G. Miller (1995). WordNet: A lexical Database for English. ACM 38 (11). pages 39-41, 1995.

[15] E.Pietriga, J-Y.Vion-Dury, and V.Quint.(2001). Vxt: a visual approach to XML transformations. In *Proceedings of the ACM Symposium on Document Engineering*, 2001.

[16] E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. In VLDB Journal, pages 10: 334-350, 2001.

[17] E. Rahm and P.A. Bernstein. On matching schema automatically. *Microsoft Research Publications*, 2001. Available at http://www.research.microsoft.com/pubs.

[18] H.Su, H.Kuno, E.A.Rundensteiner. Automating the transformation of XML Documents. In *Proceedings of the ACM Symposium on Document Engineering*, 2001.

[19] X.Tang and F. Tompa. Specifying transformations for structured documents. In *Proceedings of 4th International Worshop on Web and Databases (WebDB 2001),* pages 67-72. 2001.

[20] A. Vernet. XML transformation languages. Available at http://www.scdi.org/~avernet/misc/xml-transformation

[21] Extensible Markup Language (XML) 1.0, W3C Recommendation, 1998. Available at http://www.w3.org/TR/REC-XML

[22] XSL Transformations (XSLT) 1.0, W3C Recommendation, 1999. Available at http://www.w3.org/TR/1999/REC-xslt-19991116

[23] XML Schema Part 0: Primer, W3C Recommendation, 2001. Available at http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/

[24] XML Schema Part 2: Datatypes, W3C Recommendation, 2001. Available at http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

[25] XSLWIZ. Available at http://www.induslogic.com/products/xslwiz.html

[26] L.Xu Source Discovery and Schema Mapping for Data Integration, PhD Dissertation, July 2003