A Sequence-based Ontology Matching Approach

Alsayed Algergawy, Eike Schallehn and **Gunter Saake**[1]

Abstract. The recent growing of the Semantic Web requires the need to cope with highly semantic heterogeneities among available ontologies. Ontology matching techniques aim to tackle this problem by establishing correspondences between ontologies' elements. An intricate obstacle faces the ontology matching problem is its scalability against large number and large-scale ontologies. To tackle these challenges, in this paper, we propose a new matching framework based on Prüfer sequences. The proposed approach is applicable for matching a database of XML trees . Our approach is based on the representation of XML ontologies as sequences of labels and numbers by the Prüfer's method that constructs a one-to-one correspondence between schema ontologies and sequences. We capture ontology tree semantic information in Label Prüfer Sequences (LPS) and ontology tree structural information in Number Prüfer Sequences (NPS). Then, we develop a new structural matching algorithm exploiting both LPS and NPS. Our Experimental results demonstrate the performance benefits of the proposed approach.

1 Introduction

The Semantic Web (SW) is evolving towards an open, dynamic, distributed, and heterogenous environments. The core of the SW is ontology, which is used to represent our conceptualizations. The Semantic Web puts the onus of ontology creation on the user by providing common ontology languages such as XML, RDF(S) and OWL. However, ontologies defined by different applications usually describe their domains in different terminologies, even they cover the same domain. In order to support ontology-based information integration, tools and mechanisms are needed to resolve the semantic heterogeneity problem and align terms in different ontologies. Ontology matching plays the central role in these approaches. *Ontology matching* is the task of identifying correspondences among elements of two ontologies [5, 20, 15].

Due to the complexity of ontology/schema matching, it was mostly performed manually by a human expert. However, manual reconciliation tends to be a slow and inefficient process especially in large-scale and dynamic environments such as the Semantic Web. Therefore, the need for automatic semantic schema matching has become essential. Consequently, many ontology/schema matching systems have been developed for automating the matching process, such as Cupid [17], COMA [6], Similarity Flooding [18], LSD [7], BTreeMatch [12], Spicy [2], GLUE [8, 9], OntoBuilder [21], QOM [13], and S-Match [14]. Moreover, most of these approaches have been developed and tested using small-scale schemas. The primary focus was on matching effectiveness. Unfortunately, the effectiveness of automatic match techniques typically decreases for larger schemas. In particular, matching the complete input schemas may lead not only to long execution times, but also poor quality due to the large search space. Therefore, the need for efficient and effective algorithms has been arisen.

Recently, matching algorithms are introduced to focus on matching large-scale and large number schemas and ontologies, i.e. considering the efficiency aspect of matching algorithms, such as COMA++ [1, 11], QOM [13], Bellflower [23], and PORSCHE [22]. Most of these systems rely heavily on either rule-based approaches or learner-based approaches. In the rule-based systems, schemas to be matched are represented as schema trees or schema graphs which in turn requires traversing these trees (or graphs) many times. On the other hand, learning-based systems need much pre-effort to train its learners. As a consequence, especially in large-scale schemas and dynamic environments, matching performance declines radically.

Motivated by the above challenges and by the fact that the most prominent feature for an XML schema is its hierarchical structure, in this paper, we propose a novel approach for matching XML schemas. In particular, we develop and implement the *XPrüM* system, which consists mainly of two parts —schema preparation and schema matching. Schemas to be matched are first parsed and represented internally using rooted ordered labeled trees, called schema trees. Then, we construct a Prüfer sequence for each schema tree. Prüfer sequences construct a one-to-one correspondence between schema trees and sequences. We capture schema tree semantic information in Label Prüfer Sequences (LPS) and schema tree structural information in Number Prüfer Sequences (NPS). LPS is exploited by a linguistic matcher to compute terminological similarities between schemas elements.

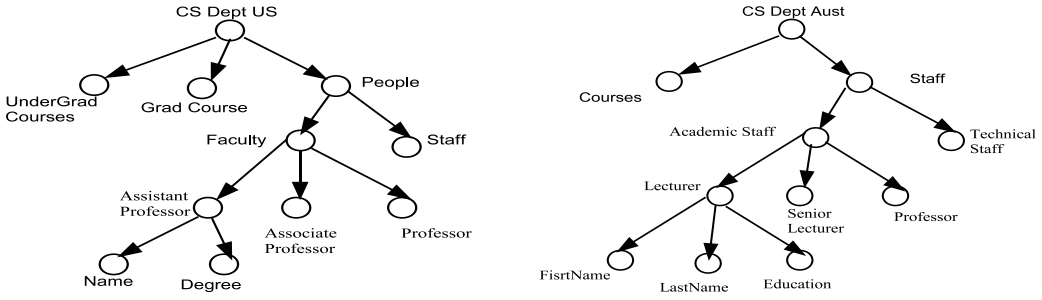
Linguistic matching techniques may provide false positive matches. Structural matching is used to correct such matches based on structural contexts of schema elements. Structural matching relies on the notion of node[2] context. In this paper, we distinguish between three types of node contexts depending on its location in the schema tree. These types are *child context, leaf context* and *ancestor context*. We exploit the number sequence representation of the schema tree to extract node contexts for each tree node in an efficient way. Then, for each node context, we apply its associated algorithm. For example, the leaf context similarity between two nodes is measured by extracting leaf context for each node as a gap vector. Then, we apply the cosine measure between two gap node vectors. Other context similarity measures are determined similarly.

By representing schema trees as Prüfer Sequences we need to traverse these trees only once to construct these sequences. Then, we develop a novel structural matching algorithm which captures semantic information existing in label sequences and structural information embedded in number sequences.

The paper is organized as follows: Section 2 introduces basic concepts and definitions. Section 3 describes our proposed approach. Section 4 presents experimental results. Section 5 gives concluding

[1] Magdeburg University, Germany, email: {alshahat, eike, saake}@ovgu.de

[2] In this paper the terms node and element are interchangeable



(a) Ontology Tree OT1 (b) Ontology Tree OT2

Figure 1: Computer science department ontologies

remarks and our proposed future work.

2 Preliminaries

The semantics conveyed by ontologies can be as simple as a database schema or as complex as the background knowledge in a knowledge base. Our proposed approach is concerning only with the first case, which gives the assumption that ontologies in this paper are represented as XML schemas.

XML schemas can be modeled as rooted, ordered, labeled trees $T = (N, E, Lab)$, called ontology trees. N is the set of tree nodes, representing different XML schema components. E is the set of edges, representing the relationships between schema components. Lab is the set of node labels, representing properties of each node. We categorize nodes into *atomic nodes*, which have no outgoing edges and represent leaf nodes and *complex nodes*, which are the internal nodes in the ontology tree.

2.1 Motivations

To the best of our knowledge no work for matching XML schemas exists which is based on *Prüer Sequences*. Most of existing matching systems rely heavily on matching schema trees/graphs. In addition, these systems perform poorly when dealing with large-scale ontologies. These shortcomings have motivated us to develop an XML schema matching system based on Prüfer Sequences, called *XPrüM*.

Our proposed system is also a schema-based approach. However, each schema tree is represented using two sequences which capture both tree semantic information and tree structure. This leads to a space efficient representation and an efficient time response when compared to the state-of-the-art systems.

2.2 A Matching Example

To describe the operation of our approach, we use the example found in [9]. It describes two XML ontologies, shown in Figure 1 (a and b), that represent organization in universities from different countries and have been widely used in the literature. The task is to discover semantic correspondences between two schemas' elements.

3 The Proposed Approach

In this section, we shall describe the core parts of the XPrüM system. As shown in Fig.2, it has two main parts: *ontology preparation* and *ontology matching*. First, ontologies are parsed using a SAX parser[3]

[3] `http://www.saxproject.org`

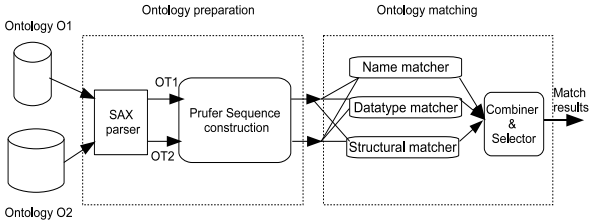


Figure 2: Matching Process Phases

and represented internally as ontology trees. Then, using the Prüfer sequence method, we extract both label sequences and number sequences. The ontology matching part discovers the set of matches between two ontologies employing both sequences.

3.1 Prüfer Sequences Construction

We now describe the tree sequence representation method, which provides a bijection between ordered, labeled trees and sequences. This representation is inspired from classical *Prüfer sequences* [19] and particularly from what is called *Consolidated Prüfer Sequence CPS* proposed in [24].

CPS of an ontology tree OT consists of two sequences, Number Prüfer Sequence *NPS* and Label Prüfer Sequence *LPS*. They are constructed by doing a *post-order traversal* that tags each node in the ontology tree with a unique traversal number. NPS is then constructed iteratively by removing the node with the smallest traversal number and appending its parent node number to the already structured partial sequence. LPS is constructed similarly but by taken the node labels of deleted nodes instead of their parent node numbers. Both NPS and LPS convey completely different but complementary information—NPS that is constructed from unique post-order traversal numbers gives wealthy tree structure information and LPS gives the labels for tree nodes. CPS representation thus provides a bijection between ordered, labeled trees and sequences. Therefore, CPS = (NPS, LPS) uniquely represents a rooted, ordered, labeled tree, where each entry in the CPS corresponds to an edge in the schema tree. For more details see [24].

Example 1. Consider ontology trees *OT1* and *OT2* shown in Figure 3, each node is associated with its *OID* and its post order number. Table 1 illustrates CPS for $OT1$ and $OT2$. For example, CPS of $OT1$ can be written as the $NPS(OT1)$= 11 11 5 5 8 8 8 10 10 11 -, and the $LPS(OT1).name$= *UnderGrad Courses, Grad Courses,*

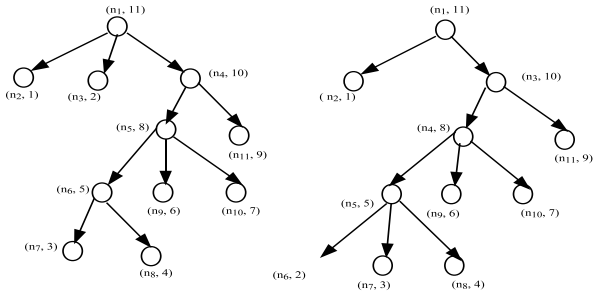


Figure 3: Nodes OIDs and corresponding post-order numbers

, Name, Degree, Assistant Professor, Associate Professor, Professor, Faculty, Staff, People, CS Dept US.

Table 1: CPS of Ontology Trees

Schema Tree ST1				Schema Tree ST2			
NPS	*LPS*			*NPS*	*LPS*		
	OID	*name*	*type/data type*		*OID*	*name*	*type/data type*
11	n_2	UnderGrad Courses	element/string	11	n_2	Courses	element/string
11	n_3	Grad Courses	element/string	5	n_6	FirstName	element/string
5	n_7	Name	element/string	5	n_7	LastName	element/string
5	n_8	Degree	element/string	5	n_8	Education	element/string
8	n_6	Assistant Professor	element/-	8	n_5	Lecturer	element/-
8	n_9	Associate Professor	element/string	8	n_9	SeniorLecturer	element/string
8	n_{10}	Professor	element/string	8	n_{10}	Professor	element/string
10	n_5	faculty	element/-	10	n_4	AcademicStaff	element/-
10	n_{11}	Staff	element/string	10	n_{11}	TecnicalStaff	element/string
11	n_4	People	element/-	11	n_3	Staff	element/-
-	n_1	CS Dept US	element/-	-	n_1	CS Dept Aust	element/-

3.1.1 CPS Properties

In the following, we list the structural properties behind CPS representation of ontology trees. If we construct a CPS=(NPS, LPS) from an ontology tree OT, we could classify these properties into:

- *Unary Properties*: for every node n_i has a postorder number *k*,
  1. *atomic node*: n_i is an atomic node iff $k \notin NPS$
  2. *complex node*: n_i is a complex node iff $k \in NPS$
  3. *root node*: n_i is the root node (n_{root}) iff $k = max(NPS)$, where *max* is a function which returns the maximum number in NPS.
- *Binary Properties*
  1. *edge relationship*: each entry $CPS_i = (k_i, LPS_i)$ represents an edge from the node whose post-order number is k_i to a node n_i = $LPS_i.OID$. This property shows both child and parent relationships. This means that the node $n_i = LPS_i.OID$ is an immediate child for the node whose post-order number k_i.
  2. *sibling relationship*: $\forall$ two entries $CPS_i = (k_i, LPS_i)$ and $CPS_j = (k_j, LPS_j)$, the two nodes $n_i = LPS_i.OID$ and $n_j = LPS_j.OID$ are two sibling nodes iff $k_i = K_j$.

3.2 Matching Algorithms

Ontology matching algorithms operate on the sequential representation of two ontology trees *ST1* and *ST2* and discover semantic correspondences between them. Generally speaking, the process of ontology matching is performed, as shown in Figure 2, in two phases—*element matchers* and *combiner & selector*.

First, a degree of similarity is computed automatically for all element pairs using the element matcher phase. Recent empirical analysis shows that there is no single dominant element matcher that performs best, regardless of the data model and application domain [10]. As a result, we should exploit different kinds of element matchers. In our approach, we make use of *name matcher*; to exploit elements'names, *datatype matcher*; to exploit elements' types/datatypes, and *structural matcher*; to exploit elements' structural contexts. After a degree of similarity is computed, how to combine different similarities from different element matchers and select top-K mappings are addressed in the second phase

3.2.1 Name Matcher

The aim of this phase is to obtain an initial matching between elements of two ontology trees based on the similarity of their names. To compute linguistic similarity between two elements'names s_1 and s_2, we use the following three similarity measures. The first one is $sim_{edit}(s_1, s_2) = \frac{max(|s_1|,|s_2|)-editDistance(s_1,s_2)}{max(|s_1|,|s_2|)}$ where *editDistance(s_1, s_2)* is the minimum number of character insertion and deletion operations needed to transform one string to the other. The second similarity measure is based on the number of different trigrams in the two strings: $sim_{tri}(s_1, s_2) = \frac{2\times|tri(s_1)\bigcap tri(s_2)|}{|tri(s_1)|+|tri(s_2)|}$ where *tri(s_1)* is the set of trigrams in s_1. The third similarity measure is based on Jaro-Winkler distance, which is given by $sim_{jaro}(s_1, s_2) = \frac{1}{3} \times (\frac{m}{|s_1|} + \frac{m}{|s_2|} - \frac{m-t}{m})$ where *m* is the number of matching characters and *t* is the number of transpositions. The linguistic matching between two ontology tree nodes is computed as the combination (weighted sum) of the above three similarity values.

3.2.2 Datatype Compatibility

We propose the use of datatype compatibility to improve initial linguistic similarity. To this end, we make use of built-in XML datatypes hierarchy [4] in order to compute datatype compatibility coefficients. Based on XML schema datatype hierarchy, we build a datatype compatibility table as the one used in [17]. After computing datatype compatibility coefficients we can adjust linguistic similarity values. The result of the above process is an adjusted linguistic similarity matrix.

3.2.3 Structural Matcher

Our structural matching algorithm is motivated by the fact that the most prominent feature in an XML schema is its hierarchical structure. This matcher is based on the node context, which is reflected by its ancestors and its descendants. The descendants of an element include both its immediate children and the leaves of the subtrees rooted at the element. The immediate children reflect its basic structure, while the leaves reflect the element's content. In this paper, as in [16, 3], we consider three kinds of node contexts depending on its position in the ontology tree:

- *The child context* of a node n_i is defined as the set of its immediate children nodes including attributes and subelements. The child context of an atomic node is an empty set. Using the edge relationship property, we could identify immediate children of a complex node and their count. The number of immediate children of a non-leaf node from the *NPS* sequence is obtained by counting its post-order traversal number in the sequence, and then we could identify these children. For example, in *Example 1, consider OT1*,

[4] http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

the post-order number of node n_4 is 10. This number repeats two times. This means that it has two immediate children $\{n_5, n_{11}\}$.

- *The leaf context* of a node n_i is defined as the set of leaf nodes of subtrees rooted at node n_i. We notice that nodes whose post-order numbers do not appear in the *NPS* sequence are atomic nodes; *CPS' atomic property*. From this notice and from the child context we could recursively obtain the leaf context for a certain node. For example, in *Example 1, consider OT1*, nodes $\{n_2, n_3, n_7, n_8, n_9, n_{10}, n_{11}\}$ are leaf nodes set. Node n_6 has two children n_7, n_8, which are leaves. Then they are the leaf context set of node n_6.
- *The ancestor context* of a node n_i is defined as the path extending from the root node to the node n_i. The ancestor of the root node is an empty path. For a non-atomic node, we obtain the ancestor context by scanning the *NPS* sequence from left to right and identifying the numbers which are greater than post-order number of the node until the first occurrence of the root node. While scanning from left to right, we ignore nodes whose post-order numbers are less than post-order numbers of already scanned nodes. For a leaf node, the ancestor context is the ancestor context of its parent union the parent itself. For example, in *Example 1, consider OT1*, the ancestor context of node n_5 (non-atomic node) is the path n_1/ n_4/ n_5. While the ancestor context of node n_9 (atomic node) is the path n_1/n_4/ n_5/ n_9.

Structural Context Similarity Algorithm Structural node context defined above relies on the notion of path and set. In order to compare two ancestor contexts, we essentially compare their corresponding paths. On the other hand, in order to compare two child contexts and/or leaf contexts, we need to compare their corresponding sets. In the following we describe how to compute the three structural context measures:

1. *Child Context Algorithm:* To obtain the child context similarity between two nodes, we compare the two child context sets for the two nodes. To this end, we first extract the child context set for each node. Second, we get the linguistic similarity between each pair of children in the two sets. Third, we select the matching pairs with maximum similarity values. And finally, we take the average of best similarity values.
2. *Leaf Context Algorithm:* Before we delve into details of computing leaf context similarity, we shall first introduce the notion of *gap* between two tree nodes.

Definition The gap between two nodes n_i and n_j in an ontology tree *OT* is defined as the difference between their post-order numbers.
To compute the leaf context similarity between two nodes, we compare their leaf context sets. To this end, first, we extract the leaf context set for each node. Second, we determine the gap between each node and its leaf context set. We call this vector the *gap vector*. Third, we apply the cosine measure between two vectors.

Example 2. For the two ontology trees shown in *Example 1*. The leaf context set of $OT1.n_1$ is *leaf_set* (n_1)=$\{n_2, n_3, n_7, n_8, n_9, n_{10}, n_{11}\}$ and the leaf context set of $OT2.n_1$ is *leaf_set* (n_1)=$\{n_2, n_6, n_7, n_8, n_9, n_{10}, n_{11}\}$. The gap vector of $ST1.n_1$ is v_1 =*gapvec(OT1.*n_1*)*=$\{10,9,8,7,5,4,2\}$ and the gap vector of $OT2.n_1$ is v_2 =*gapvec(OT2.*n_1*)*=$\{10,9,8,7,5,4,2\}$. The cosine measure CM of the two vectors gives *CM(*v_1,v_2*) =1.0*. Then the leaf context similarity between nodes $OT1.n_1$ and $OT2.n_1$ is 1.0.

3. *Ancestor Context Similarity:* The ancestor context similarity captures the similarity between two nodes based on their ancestor contexts. To compute the ancestor similarity between two nodes n_i and n_j, first we extract each ancestor context from the *CPS* sequence, say path P_i for n_i and path P_j for n_j. Second, we compare two paths. To compare two paths, we use three of four scores established in [4] and reused in [3]. These scores are combined to compute the similarity between two paths P_i and P_j *psim* as follows:

$$psim(P_i, P_j) = LCS_n(P_i, P_j) - \gamma GAPS(P_i, P_j) - \delta LD(P_i, P_j) \quad (1)$$

where γ and δ are positive parameters ranging from 0 to 1 that represent the comparative importance of each factor. The three scores are: (1)$LCS_n(P_i, P_j)$ used to measure longest common subsequences between two paths normalized by the length of the first path, (2) $GAPS(P_i, P_j)$ used to ensure that the occurrences of two paths'nodes are close to each other, and (3)$LD(P_i, P_j)$ used to give higher values to source paths whose lengths is similar to target paths.

Putting it all together: Our complete structural matching algorithm is as follows: The algorithm accepts *CPS(NPS, LPS)* for each schema tree and the linguistic similarity matrix as inputs and produces a structural similarity matrix. For each node pairs, context similarity is computed using child context, ancestor context, and leaf context. The three context values are then combined.

4 Experimental Evaluation

To implement the solution we have installed above algorithms using Java platform. We ran all our experiments on 2.4GHz Intel core2 processor with 2GB RAM running Windows XP. We shall describe the data sets used through evaluation and our experimental results.

4.1 Data Sets

We experimented with the data sets shown in Table 2. These data sets were obtained from[5] [6] [7]. We choose these data sets since they capture different characteristics in the numbers of nodes (schema size), their depth (the number of nodes nesting) and represent different application domains, see Table 2.

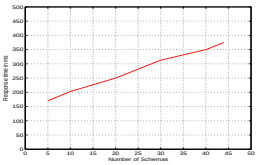
Table 2: Data set details

Domain	*No. of ontologies/nodes*	*Ontology size*
University	44/550	$<$ 1KB
XCBL	570/3500	$<$ 10 KB
OAGIS	4000/36000	$<$100 KB
OAGIS	100/65000	$>$100 KB

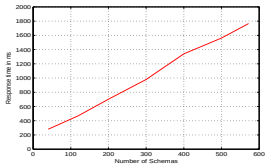
4.2 Measures for Match Performance

The *XPrüM* system considers both performance aspects —matching effectiveness and matching efficiency. However due to space limit and according the paper outline, we consider matching efficiency in more details. To this end, we use the response time as a function of the number of schemas and the number of nodes to measure matching efficiency.

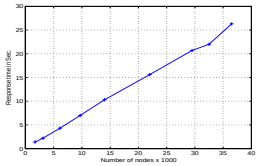
[5] http://www.cs.toronto.edu/db/clio/testSchemas.html
[6] http://www.xcbl.com
[7] http://www.oagi.org



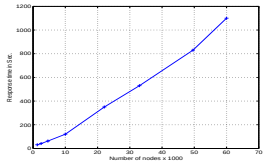
(a) Response time of University ontologies



(b) Response time of XCBL ontologies



(c) Res. time of OAGIS ontology'nodes 1



(d) Res. time of OAGIS ontology'nodes 2

Figure 4: Performance analysis of XPrüM system with real world ontologies

4.3 Experimental Results

XPrüM Quality: To show matching effectiveness of our system, ontologies shown in Figure 1 have been used. XPrüM could discover 9 true positive matches among 11 and 2 false positive matches with F-measure of 81%. Compared to COMA++ [1] which could discover only 5 true positive matches and miss 6 false negative matches with F-measure of 62%, our system demonstrates better quality.

XPrüM Efficiency: Figure 4 shows that *XPrüM* achieves high scalability across all three domains. The system could identify and discover correspondences across 44 schemas including 550 nodes from the university domain in a time of 0.4 second, while the approach needs 1.8 seconds to match 570 schemas including approximately 3500 nodes from the XCBL domain. This demonstrates that *XPrüM* is scalable with large number of schemas. To demonstrates the system scalability with large-scale schemas, we performed two set of experiments. The first one is tested with the OAGIS domain whose schemas'sizes ranging between $10KB$ and $100KB$. Figure 4(c) shows that the system needs 26 seconds to match 4000 schemas containing 36000 nodes. The second set is performed also using the OAGIS domain containing 100 schemas whose sizes are greater than $100KB$. *XPrüM* needs more than 1000 seconds to match 65000 nodes as shown in Figure 4(d).

5 CONCLUSION

With the proliferation of the Semantic Web ontologies, the development of automatic techniques for ontology matching will be crucial to their success. In this paper, we have addressed an intricate problem associated to the ontology matching problem—matching scalability. To tackle this, we have proposed and implemented the *XPrüM* system, a hybrid matching algorithm to automatically discover semantic correspondences between XML schemas. The system starts with transforming schemas into ontology trees and then constructs a consolidated Prüfer sequence *CPS* for each schema tree which construct a one-to-one correspondence between schema trees and sequences. We capture schema tree semantic information in Label Prüfer Sequences and schema tree structural information in Number Prüfer Sequences.

Experimental results have shown that *XPrüM* has a high scalability with respect to large number and large-scale schemas. Moreover, it could preserve matching quality beside its scalability. *XPrüM* has other features including: it is almost automatic; it does not make use of any external dictionary; moreover, it is independent on data mode and application domain of matched schemas.

In our ongoing work, we should consider the second aspect of semantics conveyed by ontologies, i.e. modeling ontologies using RDF(s) or OWL, to deal with background knowledge. This helps us to apply the sequence-based approach on other applications and domains such as image matching and the Web service discovery.

REFERENCES

[1] D. Aumueller, H.H. Do, S. Massmann, and E. Rahm, 'Schema and ontology matching with COMA++', in *SIGMOD Conference*, (2005).

[2] A. Bonifati, G. Mecca, A. Pappalardo, and S. Raunich, 'The spicy project: A new approach to data matching', in *SEBD*. Turkey, (2006).

[3] A. Boukottaya and C. Vanoirbeek, 'Schema matching for transforming structured documents', in *DocEng'05*, pp. 101–110, (2005).

[4] D. Carmel, N. Efraty, G. Landau, Yo. Maarek, and Y. Mass, 'An extension of the vector space model for querying xml documents via xml fragments', *SIGIR Forum*, **36**(2), (2002).

[5] L. Ding, P.Kolari, Z. Ding, S. Avancha, T. Finin, and A. Joshi, 'Using ontologies in the semantic web: A survey', in *TR-CS-05-07*, (2005).

[6] H. H. Do and E. Rahm, 'COMA- A system for flexible combination of schema matching approaches', in *VLDB 2002*, pp. 610–621, (2002).

[7] A. Doan, 'Learning to map between structured representations of datag', in *Ph.D Thesis*. Washington University, (2002).

[8] A. Doan, J. Madhavan, R. Dhamankar, P. Domingos, and A.Halevy, 'Learning to match ontologies on the semantic web', *Knowledge and Information Systems*, **12**(4), 303–319, (2003).

[9] A. Doan, J. Madhavan, P. Domingos, and A. Halevy, *Ontology matching: A machine learning approach*, Handbook on Ontologies, International Handbooks on Information Systems, 2004.

[10] C. Domshlak, A. Gal, and H. Roitman, 'Rank aggregation for automatic schema matching', *IEEE on KDE*, **19**(4), 538–553, (April 2007).

[11] C. Drumm, M. Schmitt, H.-H. Do, and E. Rahm, 'Quickmig - automatic schema matching for data migration projects', in *Proc. ACM CIKM07*. Portugal, (2007).

[12] F. Duchateau, Z. Bellahsene, and M. Roche, 'An indexing structure for automatic schema matching', in *SMDB Workshop*. Turkey, (2007).

[13] M. Ehrig and S. Staab, 'QOM - quick ontology mapping', in *International Semantic Web Conference 2004:*, pp. 683–697, (2004).

[14] F. Giunchiglia, M. Yatskevich, and P. Shvaiko, 'Semantic matching: Algorithms and implementation', *Journal on Data Semantics*, **9**, 1–38, (2007).

[15] Y. Kalfoglou and M. Schorlemme, 'Ontology mapping: the state of the art', *The Knowledge Engineering Review*, **18**(1), 1–31, (2003).

[16] M. L. Lee, L. Yang, W. Hsu, and X. Yang, 'Xclust: Clustering XML schemas for effective integration', in *CIKM'02*, pp. 63–74, (2002).

[17] J. Madhavan, P. Bernstein, and E. Rahm, 'Generic schema matching with cupid', in *VLDB 2001*, pp. 49–58. Roma, Italy, (2001).

[18] S. Melnik, H. Garcia-Molina, and 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 (ICDE'02)*, (2002).

[19] H. Prufer, 'Neuer beweis eines satzes uber permutationen', *Archiv für Mathematik und Physik*, **27**, 142–144, (1918).

[20] E. Rahm and P. Bernstein, 'A survey of approaches to automatic schema matching', *VLDB Journal*, **10**(4), 334–350, (2001).

[21] H. Roitman and A. Gal, 'Ontobuilder: Fully automatic extraction and consolidation of ontologies from Web sources using sequence semantics', in *EDBT 2006 Workshops*, (2006).

[22] K. Saleem, Z. Bellahsene, and E. Hunt, 'PORSCHE: Performance oriented schema mediation', *Accepted for publication in Information Systems Journal*, (2008).

[23] M. Smiljanic, *XML Schema Matching Balancing Efficiency and Effectiveness by means of Clustering*, Ph.D. dissertation, Twente University, 2006.

[24] S. Tatikonda, S. Parthasarathy, and M. Goyder, 'LCS-trim: Dynamic programming meets XML indexing and querying', in *VLDB'07*, pp. 63–74, (2007).