Instance Matching with COMA++

Daniel Engmann, Sabine Massmann[1]

Department of Computer Science
University of Leipzig
Johannisgasse 26
Leipzig 04103, Germany
[1]massmann@informatik.uni-leipzig.de

Abstract: Schema matching is the process of identifying semantic correspondences between schemas. COMA++ is a matching prototype which uses several characteristics of schemas to determine similarities between them, for example the names and data types of the schema elements and structural information. In this paper we propose two instance-based matchers for COMA++ to gain a further quality improvement. The features of the matchers and first results are described.

1 Introduction

Schema matching is the process of identifying semantic correspondences between schemas and the first step for data integration and transformation. Matching can be done manually but the operation is time consuming, expensive and error-prone. So an (semi-) automatic match process is needed. There exist many tools but open challenges remain, e.g. matching large ontologies and schemas.

COMA ([DR02], [RDM04]) is a generic match system that has been developed at the University of Leipzig. The prototype provides a large spectrum of matchers that can be combined in a flexible way. The enhanced version COMA++ ([Au05], [Do06]) supports matching of large real world schemas as well as ontologies using different match strategies including reuse of previous results. The graphical interface allows the user to interact and evaluate match algorithms.

Most of the matchers use schema-level data to determine a mapping. In case of opaque names, unknown synonyms and different languages these matchers cannot find all correct correspondences. We augmented COMA++ with two instance-based matchers that utilize certain constraints and linguistic approaches. As an addition these matchers apply a propagation algorithm that considers the schema elements for which no instance data is available. The import parser handles different instance sources and transforms them into the same generic data representation. Furthermore the outputs of the instance-based matchers are similarity matrices so they can be combined with other matchers. These characteristics keep the system generic.

The paper is organized as follows. Section 2 outlines the concepts of instance-based matching with COMA++ and Section 3 presents first results. In Section 4 we discuss some related work. Finally, we summarize and discuss some future work.

2 Instance Matching with COMA++

In this section we describe the concepts of instance-based matching with COMA++. In order to realize instance-based matching we had to extend the import to handle instance data. We then incorporated two different instance-based matchers into COMA++.

Figure 1 illustrates how the instance-based matchers can be applied for schema matching. First schemas and the instance data have to be parsed. Different parsers support different sources, e.g. relational databases and XML files. The parsed instance data is assigned to the generic schema representation. In the following match process matchers are executed. The instance-based matchers are a constraint-based and a content-based matcher. Each matcher generates as result a similarity matrix. This matrix contains pair-wise similarity values for the schema elements. A propagation algorithm is applied on the results of the instance-based matchers to transfer similarities from elements to their surrounding elements. Finally the similarity matrices are combined to derive a mapping. The whole match process including this combination step is described in detail in [DR02].

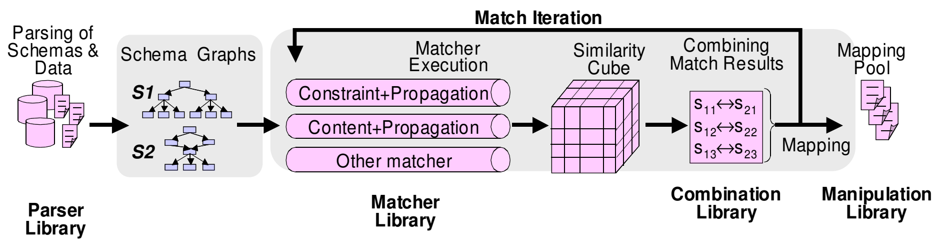
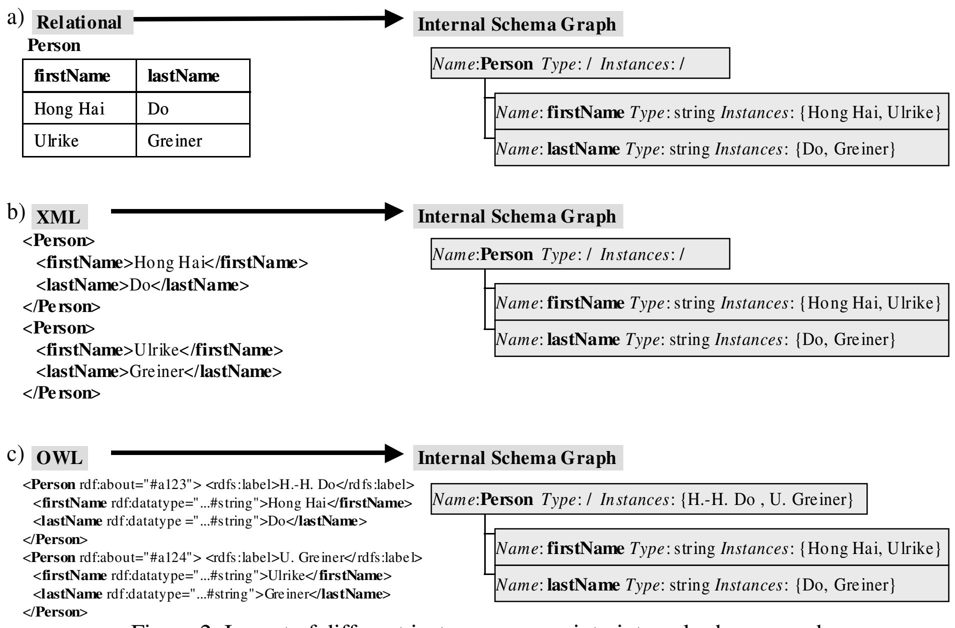


Figure 1: Matching schemas with COMA++

After the import of instance data (Subsection 2.1) we describe the constraint-based (Subsection 2.2) and the content-based matcher (Subsection 2.3). Finally, we present the propagation algorithm (Subsection 2.4).

2.1 Import of Instance Data

First we describe what we use as instance data and how. COMA++ supports multiple schema types such as XML, relational schemas and ontologies. These different schema types are parsed into a generic schema representation so that matching between different sources is possible. To keep this advantage the import of instance data is generic, too. We extended the schema representation so that each element can contain instance data. Figure 2 illustrates the import of instances from different sources into the internal schema representation. In all three cases (relational, XML, OWL) the schema graph contains three nodes in the shown example: the root *Person* and its children *firstName* and *lastName*. In addition to schema meta data like name and data type the nodes contain instances. The element node *firstName* has the instances “Hong Hai” and “Ulrike”. For OWL files the import assigns the labels given with the optional attribute rdfs:label to *Person*.


a) Relational
Internal Schema Graph
Name:Person Type: / Instances: /
Name: firstName Type: string Instances: {Hong Hai, Ulrike}
Name: lastName Type: string Instances: {Do, Greiner}
b) XML
Internal Schema Graph
Name:Person Type: / Instances: /
Name: firstName Type: string Instances: {Hong Hai, Ulrike}
Name: lastName Type: string Instances: {Do, Greiner}
c) OWL
Internal Schema Graph
Name:Person Type: / Instances: {H.-H. Do , U. Greiner}
Name: firstName Type: string Instances: {Hong Hai, Ulrike}
Name: lastName Type: string Instances: {Do, Greiner}

Person

firstName	lastName
Hong Hai	Do
Ulrike	Greiner

```
<Person>
  <firstName>Hong Hai</firstName>
  <lastName>Do</lastName>
</Person>
<Person>
  <firstName>Ulrike</firstName>
  <lastName>Greiner</lastName>
</Person>
```

```
<Person rdf:about="#a123"> <rdfs:label>H.-H. Do</rdfs:label>
  <firstName rdf:datatype="...#string">Hong Hai</firstName>
  <lastName rdf:datatype ="...#string">Do</lastName>
</Person>
<Person rdf:about="#a124"> <rdfs:label>U. Greiner</rdfs:label>
  <firstName rdf:datatype="...#string">Ulrike</firstName>
  <lastName rdf:datatype="...#string">Greiner</lastName>
</Person>
```

Figure 2: Import of different instance sources into internal schema graph

2.2 Constraint-based Matching

Before the matching the constraint-based matcher determines constraints that describe characteristics or patterns of element values. The constraints are assigned to the elements. During the match process these constraints are compared and their similarity determines the similarity of the matched elements. In case of huge sets of instance data this approach has a low effort because not the instance sets but their constraints are compared.

We distinguish between three groups of constraints:

- **General constraints** are constraints that can be determined for every instance of an element, e.g. average length and the used characters. We differentiate between letters, numeral and special characters.
  Example: Phone numbers consist of numbers and special characters like “/”, “-” and “+”.
- **Numerical constraints** determine if an instance is a number and what kind of number, e.g. positive or negative, integer or float. Furthermore an average value of the numbers and standard deviation can be calculated.
  Example: Prices are positive floats.

- **Pattern constraints** search whether all element instances follow a given pattern.
  Example: Email has the pattern *@*.* where * represents an arbitrary character string.

For determining the constraints the constraint-based matcher uses all instances of an element. In the beginning all possible constraints are considered. While going over the instances the gained knowledge reduces the number of relevant constraints that need to be checked. For example letters in instances lead to the fact that no numerical constraints have to be determined. A general pattern search is time-consuming so we restricted the search to a predefined set that covers often used patters like Email and URL. To cover more patterns we currently work on an algorithm that finds patterns based on special characters, e.g. *-*-* or */*-* where * represents an arbitrary character string.

In the match process the constraint-based matcher determines the similarity of two elements by comparing their previously identified constraints. The matcher uses a synonym table specifying the degree of compatibility between numerical respectively pattern constraints. Additionally, the similarity increases if the average length is alike.

2.3 Content-based Matching

The content-based matcher determines the similarity of two elements by executing a pair-wise comparison of instance values using a similarity function. The result is a similarity matrix with each dimension representing the instances of one element. This matrix is aggregated to one value that defines the similarity of the instance sets and thus the elements. This aggregation is done applying the following formula where n is the number of instances of e_1 and m is the number of instances of e_2 and sim is the used string similarity function:

$$\text{similarity}(e_1, e_2) = \frac{\sum_{i=1}^{n} \max_{j=1..m}(\text{sim}(inst_{e_1 i}, inst_{e_2 j})) + \sum_{j=1}^{m} \max_{i=1..n}(\text{sim}(inst_{e_1 i}, inst_{e_2 j}))}{n+m}$$

The formula uses for every instance of e_1 the highest similarity to an instance of e_2 and vice versa. These maxima are added and the number of all instances divides the resulting sum.

COMA++ supports many string similarity functions that the content-based matcher can use, e.g. trigram, edit distance or soundex. Contrary to the constraint-based approach content-based matching involves a much higher effort, because for every match between two elements up to *nxm* comparisons have to be performed. The similarity function can also be as simple as the test of equality. This reduces the costs of matching.

2.4 Propagation Algorithm

Propagation has already been addressed in previous research, e.g. [DHM05]. We are using similarity propagation because we have to deal with incomplete instance data, i.e. not every element has instances. For that reason we apply a propagation algorithm to the results of the constraint-based and content-based matchers. The *UpPropagation* algorithm propagates instance similarity from the leaves to their parents. The idea behind this is that parents that have similar children are similar too. The algorithm propagates similarity to direct parents because they have the strongest relationship.

The propagated similarity value for the parents is the average of the highest similarity value for each child. In the scenario of Figure 3 we have *Book* and *Volume* that have no instance data. Both have two children that somehow correspond to each other. The UpPropagation determines the similarity value 0.65 for *Book* and *Volume* by calculating the average of 0.9 for *title*, 0.3 for *extTitle*, 0.9 for *mainTitle*, and 0.5 for *subTitle*.

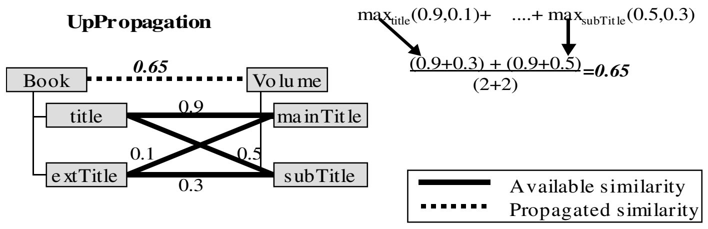


Figure 3: Scenario for similarity propagation

The UpPropagation is especially suitable for instance-based matching because normally only leaves of a schema have instance data. The algorithm is performed bottom-up and newly calculated element similarities are directly propagated to the parents of these elements. Other matchers of COMA++ do not need such a propagation because they already take surrounding elements into account, e.g. the Children matcher considers the child elements to estimate the similarity between two inner elements and the Leaves matcher uses the leaf elements for it.

3 Results for the Benchmark OAEI 2006

Ten systems including our prototype COMA++ participated in the OAEI[1] contest 2006 [MER06], which includes six tasks and the benchmark is one of them. The reference ontology (#101) contains 33 named classes, 24 object properties, 40 data properties, 56 named individuals and 20 anonymous individuals. The benchmark consists of 51 match problems in which the reference ontology has to be matched to a modified ontology. For the match problems different difficulties have been included, e.g. suppressed comments, flattened hierarchies, names replaced by random strings or suppressed instances. The match results can be compared with the provided reference alignments to determine the

[1] OAEI: Ontology Alignment Evaluation Initiative is a coordinated international initiative to forge the consensus of methods for schema matching/ontology integration. http://oaei.ontologymatching.org/

achieved precision, recall and f-measure. For all problems the same strategy and the same configuration had to be used.

We first want to evaluate the instance-based matchers and the UpPropagation in comparison to two other propagation algorithms (Subsection 3.1). Then we combine the instance-based matchers with other COMA++ matchers to get an indication of how many new correspondences have been found (Subsection 3.2).

3.1 Instance-based Matchers and Propagation Algorithms

For the evaluation of the instance-based matchers we restrict the tests to the ontologies that have at least one instance[2]. Additionally, we omit the ontology 102 because this ontology has no corresponding elements. For the remaining 39 match tasks 2966 correspondences have to be found. Only 1088 of them have instances for both elements, we call them instance-based correspondences, and thus could be found using instance-based matching. Regarding this fact we also calculated *instance-recall* that measures the found correspondences in relation to the instance-based correspondences. Table 1 shows precision[3], instance-recall, recall[4] and f-measure of different match configurations.

To evaluate the UpPropagation two other propagation strategies have been applied to the tests. For the *InstancePropagation* parent nodes inherit the instance values of their children and thus have their own instances. Then instance-based matchers are able to detect correspondences between parents as well. The *Similarity Flooding* [MGR02] takes two graphs as input and as initial mapping we use the result of the constraint-based or content-based matcher.

Used techniques	Precision	Recall	F-Measure	Instance-Recall
Constraint	0.424	0.075	0.128	0.147
InstancePropagation + Constraint	0.260	0.092	0.136	- [5]
Constraint + UpPropagation	0.467	0.091	0.153	0.169
Constraint + Similarity Flooding[6]	0.479	0.105	0.172	0.151
Content	0.997	0.353	0.521	0.904
InstancePropagation + Content	0.891	0.435	0.585	- [5]
Content + UpPropagation	0.914	0.454	0.607	0.904
Content + Similarity Flooding[6]	0.974	0.412	0.579	0.904

Table 1: Results of the instance-based matchers and different propagation algorithms

[2] Ontologies without instance data: 224, 232, 236, 237, 241, 246, 247, 301, 302, 303, 304
[3] The precision is calculated counting all correct found correspondences for all tests and all found ones and divide correct by all found.
[4] The recall is calculated counting all correct found correspondences for all tests and all expected ones and divide correct by all expected.
[5]The instance-recall has not been calculated because there are more than 1088 instance-based correspondences.
[6]We tried all four fixpoint formulas yet listed only the best result (50 iterations, fixpoint formula C).

Looking at the result we see that the constraint-based matcher behaves like a data type matcher. It finds some correspondences (and 14,7% of the instance-correspondences) but more than the half is incorrect. The reason is that most instance data contain only letters so numerical and pattern constraints are not of any use.

The content-based matcher finds 1050 correspondences and almost all are correct. The instance-recall of 0.904 shows that most of the instance-correspondences have been found. The very good precision of 0.997 depends on the fact that the instances of the ontologies are mostly equal.

Using the constraint-based matcher together with a similarity propagation algorithm such as Similarity Flooding or UpPropagation just slightly changes the result. With a poor input the output is poor as well.

Applying propagation to the content-based matcher produces a much better result – for all three algorithms. The InstancePropagation finds more correct correspondences than the Similarity Flooding but more incorrect as well. Both algorithms have almost the same f-measure of 0.58. The UpPropagation receives the highest f-measure of 0.607 and finds many correspondences (423 and 300 are correct) - more than the Similarity Flooding. The reason for that and the lower precision might be a more optimistic propagation of the similarity values.

3.2 Results of COMA++ without and with the Instance-based Matchers

In this section we determine the influence of the instance-based matcher on the quality of match results of COMA++. We ran the 39 tests of the benchmark used in Subsection 3.1 with all possible combinations of the eight most important matchers[7] from one to all matchers. Then we executed these 255 combination again – this time containing either the constraint-based or the content-based matcher as an additional matcher. Both instance-matcher use the UpPropagation which improves their result as shown in the previous subsection. The average results of the the runs are shown in Table 2.

Combinations of 8 Matcher	**Precision**	**Recall**	**F-Measure**
Without Instance-based Matcher	0.846	0.627	0.716
With Constraint+UpPropagation	0.871	0.620	0.722
With Content+UpPropagation	0.944	0.757	0.839

Table 2: Results of matcher combinations without and with instance-based matcher

The combinations without any instance-based matcher find in general 2325 correspondences and 63% of all correct correspondences. Every seventh correspondence is incorrect. Adding the constraint-based matcher just slightly changes the result. Using in addition the content-based matcher increases the precision by 10% and the recall by 13% for every matcher combination.

[7] Name, NameType, NameStat, Children, Leaves, Parents, Siblings, Comment

To see the influence of the instance-based matchers in detail we combine them with the *NameType* matcher. NameType uses the name and data type of elements to determine their similarity. Figure 4 shows the precision and recall of the different combinations of NameType, the constraint-based and the content-based matcher.

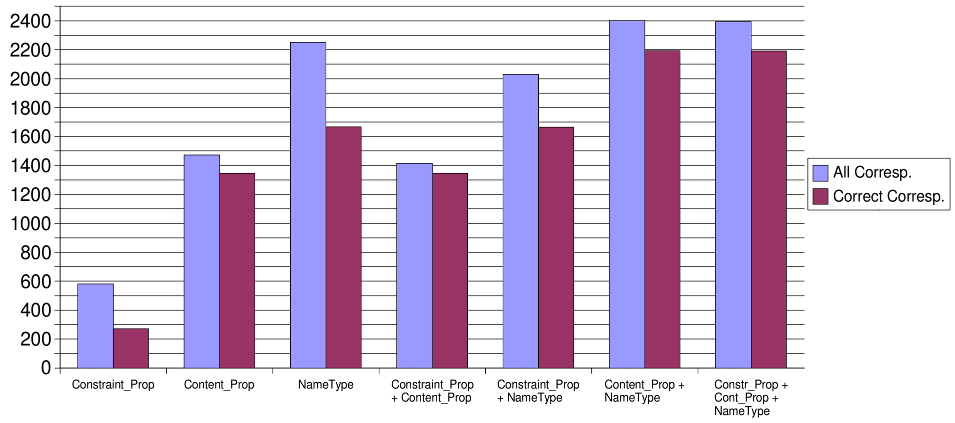


Figure 4: Results of the instance-based matchers, NameType and their combinations

The additional use of the constraint-based matcher has almost no effect. The same correct correspondences have been found and some incorrect ones less. NameType together with the content-based matcher finds 525 more correct correspondences than NameType alone due to opaque names. The precision of 0.913 is much higher than the 0.741 of NameType.

We can conclude that the content-based matcher helps mainly to detect correct correspondences that otherwise are not found because of e.g. opaque names or different structures. In addition some elements that are quite similar looking at their names or data types are identified as different due to their instance data.

4 Related Work

Various approaches have been proposed to perform schema matching ([RB01], [SE05]). In the survey [RB01] only three out of seven approaches supported instance-level matching: SemInt, LSD and Autoplex. SemInt [LC00] is a neural network-based prototype that identifies attribute correspondences in databases. It exploits up to 15 constraint-based and 5 content-based matching criteria. At the instance-level it analyses data patterns, value distributions and averages. The determination of the similarity of attributes is learned during the training process directly from the meta data that has been extracted automatically. LSD [DDH01] is a machine learning approach. At the instance-level it uses several matchers (learners) that are trained during a preprocessing step. One matcher uses Whirl and the other Naive Bayes. A third matcher searches a database to verify if a XML element is a county name. Autoplex [BM01] is based on machine learning, too, and a Naive Bayesian learner exploits instance characteristics to match attributes from a relational source to a previously constructed global schema.

The Glue system [Do04] has been developed to match ontologies and applies machine learning techniques to create the mappings between them. The Distribution Estimator is one of the system modules and uses base learners that glean many different type of information from training instances, e.g. frequencies of words and value formats.

Instance-based systems that are based on machine learning do not fully utilize the flexibility offered by the composite approach of COMA++.

An instance-based matching algorithm has been designed in the DUMAS [BN05] project. The approach detects duplicates representing the same real world object in two unaligned databases. It uses this information to automatically identify matching attributes between their schemas. The HumMer [Na06] tool for automatic data fusion assumes the databases to contain duplicates and uses DUMAS to find them.

In contrast to HumMer the instance-based matchers of COMA++ do not need duplicate objects. The constraint-based matcher determines characteristics and patterns and can not recognize duplicates. The content-based matcher needs similar instances for two elements but they can belong to different real world objects.

The two-step schema matching technique in [KN03] is instance-based and focuses on the data structure. In the first step the pair-wise attribute correlation is measured and a dependency graph is constructed. Afterwards, it runs a graph matching algorithm to find matching node pairs in the graph. This data structure is an aspect none of the instance-based matchers of COMA++ considers.

The prototype Clio [HMH01] creates a mapping between two input schemas in an interactive fashion using user feedback. The resulting mapping is a view definition over the target schema that can be executed for data transformation. The translated data can help the user to refine the mapping and check its correctness but it is not directly used by the system to do that like COMA++ does it now.

5 Conclusion

In this paper we proposed the extension of COMA++ to use instance-based matching. We described the approaches of constraint-based and content-based techniques. In addition we suggested the propagation of similarity values of elements to their parents. The instance-based matcher and the propagation algorithm have been evaluated with the benchmark of the OAEI contest '06. The comparison with a matcher that uses element names and data types showed the difference in the results.

Future work on this topic is the extension of the constraint-based component to recognize more patterns and of the content-based component to deal with enumerations. Another possibility is to sample the instances to reduce the huge amount of instance data in real world applications. Furthermore the instance-based matcher should be evaluated in other domains than ontology matching and against other instance-based tools, e.g. machine learning approaches.

References

[Au05] Aumüller, D.; Do, H.; Maßmann, S.; Rahm, E.: Schema and Ontology Matching with COMA++. In Proc. of the 2005 ACM SIGMOD Int. Conference on Management of Data. ACM Press, New York, NY, USA, 2005; pp. 906-908

[BM01] Berlin, J.; Motro, A.: Autoplex: Automated Discovery of Content forVirtual Databases. In Proc. of the 9th Int. Conference on Cooperative Information Systems. Springer-Verlag, London, UK, 2001; pp. 108-122

[BN05] Bilke, A.; Naumann, F.: Schema Matching using Duplicates. In Proc. of the 21st Int. Conference on Data Engineering (ICDE'05) - Volume 00. IEEE Computer Society, Washington, DC, USA, 2005; pp. 69-80

[DDH01] Doan, A.; Domingos, P.; Halevy, A.: Reconciling Schemas of Disparate Data Sources:A Machine-Learning Approach. In Proc. of the 2001 ACM SIGMOD Int. Conference on Management of Data. ACM Press, New York, NY, USA, 2001; pp. 509-520

[DHM05] Dong, X.; Halevy, A.; Madhavan, J.: Reference Reconciliation in Complex Information Spaces. In Proc. of the 2005 ACM SIGMOD Int. Conference on Management of Data. ACM Press, New York, NY, USA, 2005; pp. 85-96

[Do04] Doan, A.; Madhavan, J.; Domingos, P.; Halevy, A.: Ontology Matching: A Machine Learning Approach. In Handbook on Ontologies in Information Systems. Springer-Verlag, 2004; pp.

[Do06] Do, H.: Schema Matching and Mapping-based Data Integration. Verlag Dr. Müller (VDM), 2006

[DR02] Do, H.; Rahm, E.: COMA - A system for flexible combination of schema matching approaches. In Proc. 28th Int. Conference on VLDB. Springer, 2002; pp. 610-621

[HMH01] Hernández, M.; Miller, R.; Haas, L.: Clio: A Semi-Automatic Tool For Schema Mapping. In Proc. of the 2001 ACM SIGMOD Int. Conference on Management of Data. ACM Press, New York, NY, USA, 2001; pp. 607

[KN03] Kang, J.; Naughton, J.: On Schema Matching with Opaque Column Names and Data Values. In Proc. of the 2003 ACM SIGMOD Int. Conference on Management of Data. ACM Press, New York, NY, USA, 2003; pp. 205-216

[LC00] Li, W.; Clifton, C.: SEMINT: a tool for identifying attribute correspondences in heterogeneous databases using neural networks. In Data & Knowledge Engineering Volume 33, Issue 1. Elsevier Science Publishers B. V., 2000; pp. 49 - 84

[MER06] Massmann, S.; Engmann, D.; Rahm, E.: COMA++: Results for the Ontology Alignment Contest OAEI 2006. In . Int. Workshop on Ontology Matching, 2006; pp. 107-114

[MGR02] Melnik, S.; Garcia-Molina, H.; Rahm, E.: Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In . 18th Int. Conference on Data Engineering, 2002; pp. 117

[Na06] Naumann, F.; Bilke, A.; Bleiholder, J.; Weis, M.: Data Fusion in Three Steps: Resolving Inconsistencies at Schema-, Tuple-, and Value-level. In Bulletin of the Technical Committee on Data Engineering, Vol. 29 No. 2. IEEE, 2006; pp. 21-31

[RB01] Rahm, E.; Bernstein, P.: A survey of approaches to automatic schema matching. In The VLDB Journal Volume 10 , Issue 4. Springer, 2001; pp. 334-350

[RDM04] Rahm, E.; Do, H.; Maßmann, S.: Matching large XML schemas. In ACM SIGMOD Record Volume 33 , Issue 4. , 2004; pp. 26-31

[SE05] Shvaiko, P.; Euzenat, J.: A Survey of Schema-based Matching Approaches. In Journal on Data Semantics IV. Springer, 2005; pp. 146-171