Semantic Schema Matching*

Fausto Giunchiglia, Pavel Shvaiko, and Mikalai Yatskevich

University of Trento, Povo, Trento, Italy
{fausto|pavel|yatskevi}@dit.unitn.it

Abstract. We view *match* as an operator that takes two graph-like structures (e.g., XML schemas) and produces a mapping between the nodes of these graphs that correspond semantically to each other. *Semantic schema matching* is based on the two ideas: (i) we discover mappings by computing *semantic relations* (e.g., equivalence, more general); (ii) we determine semantic relations by analyzing the *meaning* (concepts, not labels) which is codified in the elements and the structures of schemas. In this paper we present basic and optimized algorithms for semantic schema matching, and we discuss their implementation within the S-Match system. We also validate the approach and evaluate S-Match against three state of the art matching systems. The results look promising, in particular for what concerns quality and performance.

1 Introduction

Match is a critical operator in many well-known metadata intensive applications, such as schema/classification/ontology integration, data warehouses, e-commerce, semantic web, etc. The match operator takes two graph-like structures and produces a mapping between the nodes of the graphs that correspond semantically to each other.

Many diverse solutions of match have been proposed so far, for example [2, 5, 7, 8, 10, 16, 17, 19]. We focus on a schema-based solution, namely a matching system exploiting only the schema information, thus not considering instances. We follow a novel approach called *semantic matching* [11]. This approach is based on the two key ideas. The first is that we calculate mappings between schema elements by computing *semantic relations* (e.g., equivalence, more generality, disjointness), instead of computing coefficients rating match quality in the [0,1] range, as it is the case in the most previous approaches, see, for example, [8, 17, 19]. The second idea is that we determine semantic relations by analyzing the *meaning* (concepts, not labels) which is codified in the elements and the structures of schemas. In particular, labels at nodes, written in natural language, are translated into propositional formulas which explicitly codify the label's intended meaning. This allows us to translate the matching problem into a propositional unsatisfiability problem, which can then be efficiently resolved using (sound and complete) state of the art propositional satisfiability (SAT) deciders, e.g., [4].

* We are grateful to John Mylopoulos and anonymous reviewers for their insightful comments on the semifinal version of the paper.

A vision for semantic matching approach and its implementation were reported in [11–13]. In contrast to that work which have been focused only on matching classifications or light-weight ontologies, this paper also considers matching XML schemas. It elaborates in more detail the element level and the structure level matching algorithms, providing a complete account of the approach. In particular, the main contributions are: (i) a new schema matching algorithm, which builds on the advances of the previous solutions at the element level by providing a library of element level matchers, and guarantees correctness and completeness of its results at the structure level; (ii) an extension of the semantic matching approach for handling attributes; (iii) the quality and performance evaluation of the implemented system called S-Match against other state of the art systems, which proves empirically the benefits of our approach.

The rest of the paper is organized as follows. Section 2 provides the related work. A basic version of the matching algorithm is articulated in its four macro steps in Section 3, while its optimizations are reported in Section 4. Section 5 discusses semantic matching with attributes. Section 6 presents a comparative evaluation. Finally, Section 7 reports conclusions and discusses the future work.

2 Related Work

At present, there exists a line of semi-automated schema matching systems, see, for instance [2, 7, 8, 10, 16, 17, 19]. A good survey and a classification of matching approaches up to 2001 is provided in [24], while an extension of its schema-based part and a user-centric classification of matching systems is provided in [25].

In particular, for individual matchers, [25] introduces the following criteria which allow for detailing further (with respect to [24]), the element and structure level of matching: *syntactic techniques* (these interpret their input as a function of their sole structures following some clearly stated algorithms, e.g., iterative fix point computation for matching graphs), *external techniques* (these exploit external resources of a domain and common knowledge, e.g., WordNet[21]), and *semantic techniques* (these use formal semantics, e.g., model-theoretic semantics, in order to interpret the input and justify their results).

The distinction between the hybrid and composite matching algorithms of [24] is useful from an architectural perspective. [25] extends this work by taking into account how the systems can be distinguished in the matter of considering the mappings and the matching task, thus representing the end-user perspective. In this respect, the following criteria are proposed: *mappings as solutions* (these systems consider the matching problem as an optimization problem and the mapping is a solution to it, e.g., [9, 19]); *mappings as theorems* (these systems rely on semantics and require the mapping to satisfy it, e.g., the approach proposed in this paper); *mappings as likeness clues* (these systems produce only reasonable indications to a user for selecting the mappings, e.g., [8, 17]).

Let us consider some recent schema-based state of the art systems in light of the above criteria.

Rondo. The Similarity Flooding (SF) [19] approach, as implemented in Rondo [20], utilizes a hybrid matching algorithm based on the ideas of similarity propagation. Schemas are presented as directed labeled graphs. The algorithm exploits only syntactic techniques at the element and structure level. It starts from the string-based comparison (common prefixes, suffixes tests) of the node's labels to obtain an initial mapping which is further refined within the fix-point computation. SF considers the mappings as a solution to a clearly stated optimization problem.

Cupid. Cupid [17] implements a hybrid matching algorithm comprising syntactic techniques at the element (e.g., common prefixes, suffixes tests) and structure level (e.g., tree matching weighted by leaves). It also exploits external resources, in particular, a precompiled thesaurus. Cupid falls into the mappings as likeness clues category.

COMA. COMA [8] is a composite schema matching system which exploits syntactic and external techniques. It provides a library of matching algorithms; a framework for combining obtained results, and a platform for the evaluation of the effectiveness of the different matchers. The matching library is extensible, it contains 6 elementary matchers, 5 hybrid matchers, and one reuse-oriented matcher. Most of them implement string-based techniques (affix, n-gram, edit distance, etc.); others share techniques with Cupid (tree matching weighted by leaves, thesauri look-up, etc.); reuse-oriented is a completely novel matcher, which tries to reuse previously obtained results for entire new schemas or for its fragments. Distinct features of COMA with respect to Cupid, are a more flexible architecture and a possibility of performing iterations in the matching process. COMA falls into the mappings as likeness clues category.

3 Semantic Matching

We focus on tree-like structures, e.g., XML schemas. Real-world schemas are seldom trees, however, there are (optimized) techniques, transforming a graph representation of a schema into a tree representation, e.g., the graph-to-tree operator of Protoplasm [3].

We call *concept of a label* the propositional formula which stands for the set of data instances that one would classify under a label it encodes. We call *concept at a node* the propositional formula which represents the set of data instances which one would classify under a node, given that it has a certain label and that it is in a certain position in a tree.

The semantic matching approach can discover the following semantic relations between the concepts of nodes of the two schemas: *equivalence* ($=$); *more general* ($\sqsupseteq$); *less general* ($\sqsubseteq$); *disjointness* ($\perp$). When none of the relations holds, the special *idk* (I don't know) relation is returned. The relations are ordered according to decreasing binding strength, i.e., from the strongest ($=$) to the weakest (*idk*), with more general and less general relations having equal binding power. The semantics of the above relations are the obvious set-theoretic semantics.

A *mapping element* is a 4-tuple $\langle ID_{ij}, n1_i, n2_j, R\rangle$, i=1,...,N1; j=1,...,N2; where ID_{ij} is a unique identifier of the given mapping element; $n1_i$ is the i-th node of the first tree, N1 is the number of nodes in the first tree; $n2_j$ is the j-th

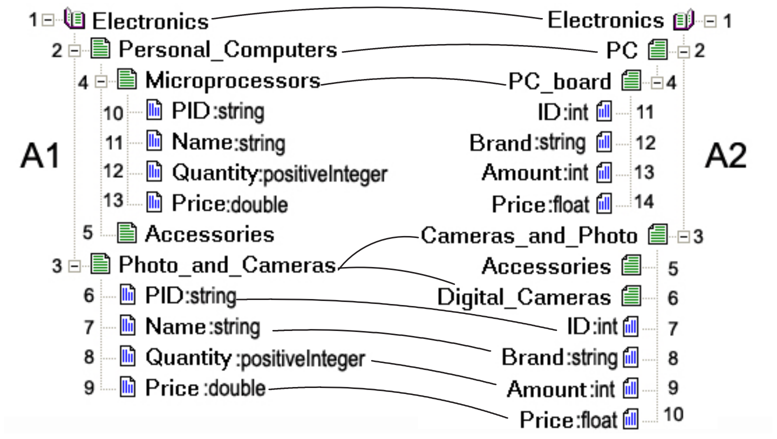

Fig. 1: Two XML schemas and some of the mappings

node of the second tree, N2 is the number of nodes in the second tree; and R specifies a semantic relation which may hold between the *concepts of nodes* $n1_i$ and $n2_j$. *Semantic matching* can then be defined as the following problem: given two trees T1, T2 compute the N1 × N2 mapping elements $\langle ID_{i,j}, n1_i, n2_j, R'\rangle$, with $n1_i \in$ T1, i=1,...,N1, $n2_j \in$ T2, j=1,...,N2 and R' the strongest semantic relation holding between the concepts of nodes $n1_i, n2_j$.

3.1 The Tree Matching Algorithm

We summarize the algorithm for semantic schema matching via a running example. We consider the two simple XML schemas shown in Figure 1.

Let us introduce some notation (see also Figure 1). Numbers are the unique identifiers of nodes. We use "C" for concepts of labels and concepts at nodes. Also we use "$C1$" and "$C2$" to distinguish between concepts of labels and concepts at nodes in tree 1 and tree 2 respectively. Thus, in A1, $C1_{Photo\_and\_Cameras}$ and $C1_3$ are, respectively, the concept of the label *Photo_and_Cameras* and the concept at node 3.

The algorithm takes as input two schemas and computes as output a set of mapping elements in four macro steps. The first two steps represent the preprocessing phase, while the third and the fourth steps are the element level and structure level matching respectively.

Step 1. For all labels *L* in the two trees, compute *concepts of labels*. We think of labels at nodes as concise descriptions of the data that is stored under the nodes. We compute the meaning of a label at a node by taking as input a label, by analyzing its real-world semantics, and by returning as output a concept of the label, C_L. Thus, for example, by writing $C_{Cameras\_and\_Photo}$ we move from the natural language ambiguous label *Cameras_and_Photo* to the concept $C_{Cameras\_and\_Photo}$, which codifies explicitly its intended meaning, namely the data which is about cameras and photo.

Technically, we codify concepts of labels as propositional logical formulas. First, we chunk labels into *tokens*, e.g., $Photo\_and\_Cameras \rightarrow \langle photo, and, cameras\rangle$;

and then, we extract *lemmas* from the tokens, e.g., $cameras \rightarrow camera$. Atomic formulas are WordNet [21] *senses* of lemmas obtained from single words (e.g., cameras) or multiwords (e.g., digital cameras). Complex formulas are built by combining atomic formulas using the connectives of set theory. For example, $C2_{Cameras\_and\_Photo} = \langle Cameras, senses_{WN\#2}\rangle \sqcup \langle Photo, senses_{WN\#1}\rangle$, where $sen-ses_{WN\#2}$ is taken to be disjunction of the two senses that WordNet attaches to *Cameras*, and similarly for *Photo*. Notice that the natural language conjunction "and" has been translated into the logical disjunction "$\sqcup$".

From now on, to simplify the presentation, we assume that the propositional formula encoding the concept of label is the label itself. We use numbers "1" and "2" as subscripts to distinguish between trees in which the given concept of label occurs. Thus, for example, $Cameras\_and\_Photo_2$ is a notational equivalent of $C2_{Cameras\_and\_Photo}$.

Step 2. For all nodes *N* in the two trees, compute *concepts of nodes*. In this step we analyze the meaning of the positions that the labels at nodes have in a tree. By doing this we *extend* concepts of labels to *concepts of nodes*, C_N. This is required to capture the knowledge residing in the structure of a tree, namely the context in which the given concept at label occurs. For example, in A2, when we write C_6 we mean the concept describing all the data instances of the electronic photography products which are digital cameras.

Technically, concepts of nodes are written in the same propositional logical language as concepts of labels. XML schemas are hierarchical structures where the path from the root to a node uniquely identifies that node (and also its meaning). Thus, following an *access criterion* semantics [14], the logical formula for a concept at node is defined as a conjunction of concepts of labels located in the path from the given node to the root. For example, $C2_6 = Electronics_2 \sqcap Cameras\_and\_Photo_2 \sqcap Digital\_Cameras_2$.

Step 3. For all pairs of labels in the two trees, compute *relations* among *concepts of labels*. Relations between concepts of labels are computed with the help of a library of element level semantic matchers. These matchers take as input two atomic concepts of labels and produce as output a semantic relation between them. Some of the them are re-implementations of the well-known matchers used in Cupid and COMA. The most important difference is that our matchers return a semantic relation (e.g., $=$, $\sqsupseteq$, $\sqsubseteq$), rather an affinity level in the [0,1] range, although sometimes using customizable thresholds.

The element level semantic matchers are briefly summarized in Table 1. The first column contains the names of the matchers. The second column lists the order in which they are executed. The third column introduces the matcher's approximation level. The relations produced by a matcher with the first approximation level are always correct. For example, $name \sqsupseteq brand$ returned by *WordNet*. In fact, according to WordNet *name* is a hypernym (superordinate word) to *brand*. Notice that in WordNet *name* has 15 senses and *brand* has 9 senses. We use some sense filtering techniques to discard the irrelevant senses for the given context, see [18] for details. The relations produced by a matcher with the second approximation level are likely to be correct (e.g., $net = network$,

Table 1: Element level semantic matchers.

Matcher name	**Execution order**	**Approximation level**	**Matcher type**	**Schema info**
Prefix	2	2	String-based	Labels
Suffix	3	2	String-based	Labels
Edit distance	4	2	String-based	Labels
Ngram	5	2	String-based	Labels
Text corpus	12	3	String-based	Labels + corpus
WordNet	1	1	Sense-based	WordNet senses
Hierarchy distance	6	3	Sense-based	WordNet senses
WordNet gloss	7	3	Gloss-based	WordNet senses
Extended WordNet gloss	8	3	Gloss-based	WordNet senses
Gloss comparison	9	3	Gloss-based	WordNet senses
Extended gloss comparison	10	3	Gloss-based	WordNet senses
Extended semantic gloss comparison	11	3	Gloss-based	WordNet senses

but *hot* = *hotel* by *Suffix*). The relations produced by a matcher with the third approximation level depend heavily on the context of the matching task (e.g., *cat* = *dog* by *Extended gloss comparison* in the sense that they are both *pets*). Notice that matchers are executed following the order of increasing approximation. The fourth column reports the matcher's type, while the fifth column describes the matcher's input.

As from Table 1, we have three main categories of matchers. *String-based* matchers have two labels as input (with exception of *Text corpus* which takes in input also a text corpus). These compute only equivalence relations (e.g., equivalence holds if the weighted *distance* between the input strings is lower than a threshold). *Sense-based* matchers have two WordNet senses in input. The *WordNet* matcher computes equivalence, more/less general, and disjointness relations; while *Hierarchy distance* computes only the equivalence relation. *Gloss-based* matchers also have two WordNet senses as input, however they exploit techniques based on comparison of textual definitions (*glosses*) of the words whose senses are taken in input. These compute, depending on a particular matcher, the equivalence, more/less general relations. The result of step 3 is a matrix of the relations holding between concepts of labels. A part of this matrix for the example of Figure 1 is shown in Table 2.

Table 2: The matrix of semantic relations holding between concepts of labels.

	$Cameras_2$	$Photo_2$	$Digital\_Cameras_2$
$Photo_1$	*idk*	$=$	*idk*
$Cameras_1$	$=$	*idk*	$\sqsupseteq$

Step 4. For all pairs of nodes in the two trees, compute *relations* among *concepts of nodes*. During this step, we initially reformulate the tree matching problem into a set of node matching problems (one problem for each pair of nodes). Finally, we translate each node matching problem into a propositional validity problem. Let us discuss in detail the tree matching algorithm, see Algorithm 1 for the pseudo-code.

In line 6, the treeMatch function takes two trees of Nodes (source and target) in input. It starts from the element level matching. Thus, in line 11, the matrix

Algorithm 1 The tree matching algorithm

```
1: Node: struct of
2:         int nodeId;
3:         String label;
4:         String cLabel;
5:         String cNode;
6: String[ ][ ] treeMatch(Tree of Nodes source, target)
7: Node sourceNode, targetNode;
8: String[ ][ ] cLabsMatrix, cNodesMatrix, relMatrix;
9: String axioms, context_1, context_2;
10: int i,j;
11: cLabsMatrix = fillCLabMatrix(source, target);
12: for each sourceNode ∈ source do
13:   i = getNodeId(sourceNode);
14:   context_1 = getCnodeFormula(sourceNode);
15:   for each targetNode ∈ target do
16:     j = getNodeId(targetNode);
17:     context_2 = getCnodeFormula(targetNode);
18:     relMatrix = extractRelMatrix(cLabMatrix, sourceNode, targetNode);
19:     axioms = mkAxioms(relMatrix);
20:     cNodesMatrix[i][j] = nodeMatch(axioms, context_1, context_2);
21:   end for
22: end for
23: return cNodesMatrix;
```

of relations holding between concepts of labels (cLabsMatrix) is populated by the fillCLabsMatrix function which uses the library of element level matchers. We run two loops over all the nodes of source and target trees in lines 12-22 and 15-21 in order to formulate all our node matching problems. Then, for each node matching problem we take a pair of propositional formulas encoding concepts of nodes and relevant relations holding between concepts of labels using the getCnodeFormula and extractRelMatrix functions respectively. The former are memorized as context$_1$ and context$_2$ in lines 14 and 17. The latter are memorized in relMatrix in line 18. In order to reason about relations between concepts of nodes, we build the premises (axioms) in line 19. These are a conjunction of the concepts of labels which are related in relMatrix. For example, the task of matching $C1_3$ and $C2_6$, requires the following axioms: $(Electronics_1 = Electronics_2) \sqcap (Cameras_1 = Cameras_2) \sqcap (Photo_1 = Photo_2) \sqcap (Cameras_1 \sqsupseteq Digital\_Cameras_2)$. Finally, in line 20, the relations holding between the concepts of nodes are calculated by nodeMatch and are reported in line 23 as a bidimensional array (cNodesMatrix). A part of this matrix for the example of Figure 1 is shown in Table 3.

Table 3: The matrix of semantic relations holding between concepts of nodes (the matching result).

	$C2_1$	$C2_2$	$C2_3$	$C2_4$	$C2_5$	$C2_6$
$C1_3$	$\sqsubseteq$	idk	$=$	idk	$\sqsupseteq$	$\sqsupseteq$

3.2 The Node Matching Algorithm

We translate the node matching problem into a propositional validity problem. Semantic relations are translated into propositional connectives using the rules described in Table 4 (second column). The criterion for determining whether a relation holds between concepts of nodes is the fact that it is entailed by the premises. Thus, we have to prove that the following formula:

$$axioms \longrightarrow rel(context_1, context_2) \tag{1}$$

is valid, namely that it is *true* for all the truth assignments of all the propositional variables occurring in it. *axioms*, $context_1$, and $context_2$ are as defined in the tree matching algorithm. *rel* is the semantic relation that we want to prove holding between $context_1$ and $context_2$. The algorithm checks for the validity of formula (1) by proving that its negation, i.e., formula (2), is unsatisfiable.

$$axioms \wedge \neg rel(context_1, context_2) \tag{2}$$

Table 4 (third column) describes how formula (2) is translated before testing each semantic relation. Notice that (2) is in Conjunctive Normal Form (CNF), namely it is a conjunction of disjunctions of atomic formulas. In this case we assume that atomic formulas never occur negated, following what is common practice in building labels of, e.g., XML schemas. Also, notice that $a = b$ iff both $a \sqsubseteq b$ and $b \sqsubseteq a$ hold, therefore we do not need to test the equivalence relation separately.

Table 4: The relationship between semantic relations and propositional formulas.

$rel(a,b)$	Translation of $rel(a,b)$ into propositional logic	Translation of formula (2) into Conjunctive Normal Form
$a = b$	$a \leftrightarrow b$	N/A
$a \sqsubseteq b$	$a \rightarrow b$	$axioms \wedge context_1 \wedge \neg context_2$
$a \sqsupseteq b$	$b \rightarrow a$	$axioms \wedge context_2 \wedge \neg context_1$
$a \perp b$	$\neg(a \wedge b)$	$axioms \wedge context_1 \wedge context_2$

The pseudo-code of a basic solution for the node matching algorithm is provided in Algorithm 2. Let us analyze it in detail. In lines 110 and 140, the

Algorithm 2 The node matching algorithm

```
100. String nodeMatch(String axioms, context$_1$, context$_2$)
110. String formula = And(axioms, context$_1$, Not(context$_2$));
120. String formulaInCNF = convertToCNF(formula);
130. boolean isLG = isUnsatisfiable(formulaInCNF);
140. formula = And(axioms, Not(context$_1$), context$_2$);
150. formulaInCNF = convertToCNF(formula);
160. boolean isMG = isUnsatisfiable(formulaInCNF);
170. if(isMG && isLG) then
180.    return "=";
190. endif
200. if (isLG) then
210.    return "$\sqsubseteq$";
220. endif
230. if (isMG) then
240.    return "$\sqsupseteq$";
250. endif
260. formula = And(axioms, context$_1$, context$_2$);
270. formulaInCNF = convertToCNF(formula);
280. boolean isOpposite = isUnsatisfiable(formulaInCNF);
290. if (isOpposite) then
300.    return "$\perp$";
310. else
320.    return "idk";
330. endif
```

nodeMatch function constructs the formulas for testing the less general and more general relations. In lines 120 and 150, it converts them into CNF, while in lines 130 and 160, it checks formulas in CNF for unsatisfiability. If both relations hold, then the equivalence relation is returned (line 180). Finally, the same procedure is repeated for the disjointness relation. If all the tests fail, the idk relation is returned (line 320).

In order to check the unsatisfiability of a propositional formula in a basic version of our NodeMatch algorithm we use the standard DPLL-based SAT solver [4, 6]. From the example in Figure 1, trying to prove that $C2_6$ is less general than $C1_3$, requires constructing formula (3), which turns out to be unsatisfiable, and therefore, the less generality holds.

$$\begin{array}{l} ((Electronics_1 \leftrightarrow Electronics_2) \wedge (Photo_1 \leftrightarrow Photo_2) \wedge \\ (Cameras_1 \leftrightarrow Cameras_2) \wedge (Digital\_Cameras_2 \rightarrow Cameras_1)) \wedge \\ (Electronics_2 \wedge (Cameras_2 \vee Photo_2) \wedge Digital\_Cameras_2) \wedge \neg \\ (Electronics_1 \wedge (Photo_1 \vee Cameras_1)) \end{array} \tag{3}$$

4 Efficient Semantic Matching

In this section we present a set of optimizations for the node matching algorithm. In particular, we show, that when dealing with *conjunctive concepts at nodes*, i.e., the concept of node is a conjunction (e.g., $C1_2 = Electronics_1 \wedge Personal\_Computers_1$), these node matching tasks can be solved in linear time. When we have *disjunctive concepts at nodes*, i.e., the concept of node contains both conjunctions and disjunctions in any order (e.g., $C2_6 = Electronics_2 \wedge (Cameras_2 \vee Photos_2) \wedge Digital\_Cameras_2$), we use techniques avoiding the exponential space explosion which arises due to the conversion of disjunctive formulas into CNF. This modification is required since all state of the art SAT deciders take CNF formulas in input.

4.1 Conjunctive concepts at nodes

Let us make some observations with respect to Table 4. The first observation is that *axioms* remains the same for all the tests, and it contains only clauses with two variables. In the worst case, it contains $2 \cdot n_1 \cdot n_2$ clauses, where n_1 and n_2 are the number of atomic concepts of labels occurred in $context_1$ and $context_2$ respectively. The second observation is that the formulas for less and more generality tests are very similar and they differ only in the negated context formula (e.g., in the less generality test $context_2$ is negated). This means that formula (1) contains one clause with n_2 variables plus n_1 clauses with one variable. In the case of disjointness test $context_1$ and $context_2$ are not negated. Therefore, formula (1) contains $n_1 + n_2$ clauses with one variable. For lack of space, let us only consider tests for more/less general relations.

Less and more generality tests. Using the above observations, formula (1), with respect to the less/more generality tests, can be represented as follows:

$$\overbrace{\bigwedge_{0}^{n*m} (\neg A_s \vee B_t) \wedge \bigwedge_{0}^{n*m} (A_k \vee \neg B_l) \wedge \bigwedge_{0}^{n*m} (\neg A_p \vee \neg B_r)}^{axioms} \wedge \overbrace{\bigwedge_{i=1}^{n} A_i}^{context_1} \wedge \overbrace{\bigvee_{j=1}^{m} \neg B_j}^{\neg context_2} \tag{4}$$

where n is the number of variables in $context_1$, m is the number of variables in $context_2$. The Ai's belong to $context_1$, and the Bj's belong to $context_2$. s, k, p are in the [0..n] range, while t, l, r are in the [0..m] range. Axioms can be empty. Formula (4) is composed of clauses with one or two variables plus one clause with possibly more variables (the clause corresponding to the negated context). Notice that formula (4) is *Horn*, i.e., each clause contains at most one positive literal. Therefore, its satisfiability can be decided in linear time by the *unit resolution rule*. DPLL-based SAT solvers in this case require quadratic time. In order to understand how the linear time algorithm works, let us suppose that we want to check if $C1_4$ is less general than $C2_4$. Formula (4) in this case is as follows:

$$\begin{aligned}&((\neg \boldsymbol{Electronics_1} \vee Electronics_2) \wedge (\boldsymbol{Electronics_1} \vee \neg Electronics_2) \wedge \\ &(\neg \boldsymbol{Personal\_Computers_1} \vee PC_2) \wedge (\boldsymbol{Personal\_Computers_1} \vee \neg PC_2) \wedge \\ &(\neg \boldsymbol{Microprocessors_1} \vee \neg PC\_board_2)) \wedge \\ &\boldsymbol{Electronics_1} \wedge \boldsymbol{Personal\_Computers_1} \wedge \boldsymbol{Microprocessors_1} \wedge \\ &(\neg Electronics_2 \vee \neg PC_2 \vee \neg PC\_board_2)\end{aligned} \tag{5}$$

where the variables from $context_1$ are written in bold. First, we assign true to all the unit clauses occurring in (5) positively. Notice that these are all and only the clauses in $context_1$, namely, $\boldsymbol{Electronics_1}$, $\boldsymbol{Personal\_Computers_1}$, and $\boldsymbol{Microprocessors_1}$. This allows us to discard the clauses where variables from $context_1$ occur positively, namely, $(\boldsymbol{Electronics_1} \vee \neg Electronics_2)$ and $(\boldsymbol{Personal\_Computers_1} \vee \neg PC_2)$. Thus, the resulting formula is as follows:

$$\begin{aligned}&(Electronics_2 \wedge PC_2 \wedge \neg PC\_board_2) \wedge \\ &(\neg Electronics_2 \vee \neg PC_2 \vee \neg PC\_board_2)\end{aligned} \tag{6}$$

Formula (6) does not contain any variable from $context_1$. By assigning true to $Electronics_2$ and false to PC_board_2 we do not determine a contradiction, and therefore, (6) is satisfiable.

For formula (6) to be unsatisfiable, all the variables occurring in the negation of $context_2$, namely, $(\neg Electronics_2 \vee \neg PC_2 \vee \neg PC\_board_2)$ should occur positively in the unit clauses obtained after resolving *axioms* with the unit clauses in $context_1$, namely, $Electronics_2$ and PC_2. For this to happen, for any Bj there must be a clause of the form $\neg Ai \vee Bj$ in *axioms*. Formulas of the form $\neg Ai \vee Bj$ occur in (4) iff we have the axioms of type $Ai = Bj$ and $Ai \sqsubseteq Bj$. These considerations suggest the following algorithm for testing satisfiability:

– *Step 1.* Create an array of size m. Each entry in the array stands for one Bj in (4).
– *Step 2.* For each axiom of type $Ai = Bj$ and $Ai \sqsubseteq Bj$ mark the corresponding Bj.
– *Step 3.* If all the Bj's are marked, then the formula is unsatisfiable.

Thus, nodeMatch can be optimized by using Algorithm 3. The numbers on the left indicate where the new code must be positioned in Algorithm 2. fastHornUnsatCheck implements the three steps above. Step 1 is performed in lines 402 and 403. In lines 404-409, a loop on axioms implements Step 2. The final loop in lines 410-416 implements Step 3.

Algorithm 3 Optimizations: less/more generality tests

```
101. if (context$_1$ and context$_2$ are conjunctive) then
102.    isLG = fastHornUnsatCheck(context$_1$, axioms, "$\sqsubseteq$");
103.    isMG = fastHornUnsatCheck(context$_2$, axioms, "$\sqsupseteq$");
104. endif

401. boolean fastHornUnsatCheck(String context, axioms, rel)
402. int m = getNumOfVar(String context);
403. boolean array[m];
404. for each axiom $\in$ axioms do
405.    if (getAType(axiom) = {"=" $\|$ rel}) then
406.       int j = getNumberOfSecondVariable(axiom);
407.       array[j] = true;
408.    endif
409. endfor
410. for (i=0; i<m; i++) do
411.    if (!array[i]) then
412.       return false;
413.    else
414.       return true;
415.    endif
416. endfor
```

4.2 Disjunctive concepts at nodes

Now, we allow for the concepts of nodes to contain conjunctions and disjunctions in any order. As from Table 4, *axioms* is the same for all the tests. However, $context_1$ and $context_2$ may contain any number of disjunctions. Some of them are coming from the concepts of labels, while others may appear from the negated $context_1$ or $context_2$ (e.g., see less/more generality tests). With disjunctive concepts at nodes, formula (1) is a full propositional formula, and hence, no hypothesis can be made on its structure. Thus, its satisfiability must be tested by using a standard SAT decider.

In order to avoid the exponential space explosion, which may arise when converting formula (1) into CNF, we apply a set of structure preserving transformations [23]. The main idea is to replace disjunctions occurring in the original formula with newly introduced variables and to explicitly state that these variables imply the subformulas they substitute. Therefore, the size of the propositional formula in CNF grows linearly with respect to the number of disjunctions in the original formula. Thus, nodeMatch should be optimized by replacing all the calls to convertToCNF with calls to optimizedConvertToCNF.

5 Semantic Matching with Attributes

XML elements may have attributes. Attributes are $\langle attribute - name, type \rangle$ pairs associated with elements. Names for the attributes are usually chosen such that they describe the roles played by the domains in order to ease distinguishing between their different uses. For example, in A1, the attributes PID and $Name$ are defined on the same domain *string*, but their intended uses are the internal (unique) product identification and representation of the official product's names respectively. There are no strict rules telling us when data should be represented as elements, or as attributes, and obviously there is always more than one way to encode the same data. For example, in A1, $PIDs$ are encoded as *string*s, while in A2, IDs are encoded as *int*s. However, both attributes serve for the same purpose of the unique product's identification. These observations suggest two possible ways to perform semantic matching with attributes: (i) taking into account datatypes, and (ii) ignoring datatypes.

The semantic matching approach is based on the idea of matching concepts, not their direct physical implementations, such as elements or attributes. If names of attributes and elements are abstract entities, therefore, they allow for building arbitrary concepts out of them. Instead, datatypes, being concrete entities, are limited in this sense. Thus, a plausible way to match attributes using the semantic matching approach is to discard the information about datatypes. In order to support this claim, let us consider both cases in turn.

5.1 Exploiting datatypes

In order to reason with datatypes we have created a *datatype ontology*, $\boldsymbol{O_D}$, specified in OWL [26]. It describes the most often used XML schema built-in datatypes and relations between them. The backbone taxonomy of $\boldsymbol{O_D}$ is based on the following rule: *the is-a relationship holds between two datatypes iff their value spaces are related by set inclusion.* Some examples of axioms of $\boldsymbol{O_D}$ are: $\mathsf{float} \sqsubseteq \mathsf{double}$, $\mathsf{int} \perp \mathsf{string}$, $\mathsf{anyURI} \sqsubseteq \mathsf{string}$, and so on. Let us discuss how datatypes are plugged within the four macro steps of the algorithm.

Steps 1,2. Compute concepts of labels and nodes. In order to handle attributes, we extend propositional logics with the quantification construct and datatypes. Thus, we compute concepts of labels and concepts of nodes as formulas in description logics (DL), in particular, using $\mathcal{ALC}(\mathcal{D})$ [22]. For example, $C1_7$, namely, the concept of node describing all the string data instances which are the names of electronic photography products is encoded as $Electronics_1 \sqcap (Photo_1 \sqcup Cameras_1) \sqcap \exists Name_1.string$.

Step 3. Compute relations among concepts of labels. In this step we extend our library of element level matchers by adding a *Datatype* matcher. It takes as input two datatypes, it queries $\boldsymbol{O_D}$ and retrieves a semantic relation between them. For example, from axioms of $\boldsymbol{O_D}$, the *Datatype* matcher can learn that $\mathsf{float} \sqsubseteq \mathsf{double}$, and so on.

Step 4. Compute relations among concepts of nodes. In the case of attributes, the node matching problem is translated into a DL formula, which is further checked for its unsatisfiability using sound and complete procedures. Notice that in this case we have to test for modal satisfiability, not propositional satisfiability. The system we use is Racer [15]. From the example in Figure 1, trying to prove that $C2_{10}$ is less general than $C1_9$, requires constructing the following formula:

$$\begin{aligned}
&((Electronics_1{=}Electronics_2)\sqcap(Photo_1{=}Photo_2)\sqcap\\
&(Cameras_1{=}Cameras_2)\sqcap(Price_1{=}Price_2)\sqcap(float\sqsubseteq double))\sqcap\\
&(Electronics_2\sqcap(Cameras_2\sqcup Photo_2)\sqcap\exists Price_2.float)\sqcap\neg\\
&(Electronics_1\sqcap(Photo_1\sqcup Cameras_1)\sqcap\exists Price_1.double)
\end{aligned} \tag{7}$$

It turns out that formula (7) is unsatisfiable. Therefore, $C2_{10}$ is less general than $C1_9$. However, this result is not what the user expects. In fact, both $C1_9$ and $C2_{10}$ describe prices of electronic products, which are photo cameras. The storage format of *prices* in A1 and A2 (i.e., *double* and *float* respectively) is not an issue at this level of detail.

Thus, another semantic solution of taking into account datatypes would be to build abstractions out of the datatypes, e.g., float, double, decimal should be abstracted to type numeric, while token, name, normalizedString should be abstracted to type string, and so on. However, even such abstractions do not improve the situation, since we may have, for example, an *ID* of type numeric in the first schema, and a conceptually equivalent *ID*, but of type string, in the second schema. If we continue building such abstractions, we result in having that numeric is equivalent to string in the sense that they are both datatypes.

The last observation suggests that for the semantic matching approach to be correct, we should assume, that all the datatypes are equivalent between each other. Technically, in order to implement this assumption, we should add corresponding axioms (e.g., float = double) to the premises of formula (1). On the one hand, with respect to the case of not considering datatypes (see, Section 5.2), such axioms do not affect the matching result from the quality viewpoint. On the other hand, datatypes make the matching problem computationally more expensive by requiring to handle the quantification construct.

5.2 Ignoring datatypes

In this case, information about datatypes is discarded. For example, $\langle Name, string \rangle$ becomes $Name$. Then, the semantic matching algorithm builds concepts of labels out of attribute's names in the same way as it does in the case of element's names, and so on. Finally, it computes mappings using the optimized algorithm of Section 4. A part of the cNodesMatrix with relations holding between attributes for the example of Figure 1 is presented in Table 5. Notice that this solution allows us for a mapping's computation not only between the attributes, but also between attributes and elements.

Table 5: Attributes: the matrix of semantic relations holding between concepts of nodes (the matching result).

	$C2_7$	$C2_8$	$C2_9$	$C2_{10}$
$C1_6$	$=$	idk	idk	idk
$C1_7$	idk	$\sqsupseteq$	idk	idk
$C1_8$	idk	idk	$=$	idk
$C1_9$	idk	idk	idk	$=$

The task of determining mappings typically represents a first step towards the ultimate goal of, for example, data translation, query mediation, data integration, agent communication, and so on. Although information about datatypes will be necessary for accomplishing an ultimate goal, we do not discuss this issue any further since in this paper we concentrate only on the mappings discovery task.

6 Comparative Evaluation

In this section, we present the quality and performance evaluation of the matching system we have implemented, called S-Match. In particular, we validate basic and optimized versions of our system, called (S-Match$_b$) and (S-Match$_o$) respectively, and evaluate them against three state of the art matchers, namely Cupid [17], COMA [8][1], and SF [19] as implemented in Rondo [20]. All the systems under consideration are fairly comparable because they are all schema-based. They differ in the specific matching techniques they use and in how they compute mappings.

In our evaluation we have used five pairs of schemas: two artificial examples, a pair of product schemas (our running example, i.e., A1 vs. A2), a pair of purchase order schemas (CIDX vs. Excel), and a pair of parts of web directories (Google vs. Looksmart). Table 6 provides some indicators of the complexity of the test cases[2]. As match quality measures we have used the following indicators: *precision*, *recall*, *overall*, *F-measure* (see, [8]). *Precision* varies in the [0,1] range; the higher the value, the smaller is the set of wrong mappings (false positives) which have been computed. *Precision* is a correctness measure. *Recall* varies in the [0,1] range; the higher the value, the smaller is the set of correct mappings (true positives) which have not been found. *Recall* is a completeness measure. *F-measure* varies in the [0,1] range. The version computed here is the harmonic mean of *precision* and *recall*. It is a global measure of the matching quality, growing with it. *Overall* is an estimate of the post-match efforts needed for adding false negatives and removing false positives. *Overall* varies in the [-1,1] range; the higher it is, the less post-match efforts are needed. As a performance measure we have used *time*. It estimates how fast systems are when producing mappings fully automatically.

To provide a ground for evaluating the quality of match results, initially, the schemas have been manually matched to produce expert mappings. Then, the results computed by systems have been compared with expert mappings. There are three further observations that ensure a fair comparative study. The first observation is that Cupid, COMA, and Rondo can discover only the mappings which express similarity between schema elements. Instead, S-Match, among the others, discovers the disjointness relation which can be interpreted as strong dissimilarity in terms of the other systems under consideration. Therefore, we did not take into account the disjointness relations (e.g., $\langle ID_{4,4}, C1_4, C2_4, \perp\rangle$) when specifying the expert mappings. The second observation is that, since S-Match returns a matrix of relations, while all the other systems return a list of the best mappings, we used some filtering rules. More precisely we have the following two rules: (i) discard all the mappings where the relation is idk; (ii) return always the *core* relations, and discard relations whose existence is implied

[1] We thank Phil Bernstein, Hong Hai Do, and Erhard Rahm for providing us with Cupid and COMA. In the evaluation we use the version of COMA described in [8]. A newer version of the system COMA++ exists but we do not have it.

[2] Source files, description of the test cases, and expert mappings can be found at http://www.dit.unitn.it/∼accord/, experiments section.

Table 6: Some indicators of the complexity of the test cases

	#nodes	**max depth**	**#labels per tree**	**concepts of nodes**
Artificial Example #1	250/500	16/15	250/500	conjunctive
Artificial Example #2	10/10	10/10	30/30	disjunctive
A1 vs. A2	13/14	4/4	14/15	conjunctive & disjunctive
CIDX vs. Excel	34/39	3/3	56/58	conjunctive & disjunctive
Google vs. Looksmart	706/1081	11/16	1048/1715	conjunctive & disjunctive

by the core relations. For example, $\langle ID_{3,3}, C1_3, C2_3, =\rangle$ should be returned, while $\langle ID_{3,5}, C1_3, C2_5, \sqsupseteq\rangle$ should be discarded. Finally, whether S-Match returns the equivalence or subsumption relations does not affect the quality indicators. What only matters is the presence of the mappings standing for those relations.

In our experiments each test has two degrees of freedom: *directionality* and *use of oracles*. By directionality we mean here the direction in which mappings have been computed: from the first schema to the second one (forward direction), or vice versa (backward direction). For lack of space we report only the best results obtained with respect to directionality, and use of oracles allowed. We were not able to plug a thesaurus in Rondo, since the version we have is standalone, and it does not support the use of external thesauri. Thesauri of S-Match, Cupid, and COMA were expanded with terms necessary for a fair competition (e.g., expanding *uom* into *unitOfMeasure*, a complete list is available at the URL in footnote 2).

All the tests have been performed on a P4-1700, 512 MB of RAM, Windows XP, with no applications running but a single matching system. Notice, that the systems were limited to allocate no more than 512 MB of main memory. Also, all the tuning parameters (e.g., thresholds, strategies) of the systems were taken by default (e.g., for COMA we used NamePath and Leaves matchers combined in the Average strategy) for all the tests.

6.1 Test Cases

Let us discuss artificially designed problems in order to evaluate the performance of S-Match$_o$ in *ideal* conditions, namely when we have only conjunctive or disjunctive concepts of nodes. Since examples are artificial and our optimizations address only efficiency, not quality, we analyze here only the performance *time* of the systems, see, Figure 2 (Artificial Examples).

On the example with conjunctive concepts at nodes (Artificial Example #1), COMA performs 4 times slower and 15 times slower than S-Match$_b$ and S-Match$_o$ respectively. S-Match$_o$ runs around 29% faster than Rondo. Instead, Cupid runs out of memory.

On the example with disjunctive concepts at nodes (Artificial Example #2), S-Match$_o$ works around 4 orders of magnitude faster than S-Match$_b$, around 5 times faster than COMA, 1.6 times faster than Cupid, and as fast as Rondo. The significant improvement of our optimized algorithm can be explained by considering that S-Match$_b$ does not control the exponential space explosion on such matching problems. In fact, the biggest formula in this case consists of about 118000 clauses. The optimization introduced in the Section 4.2 reduces this number to approximately 20-30 clauses.

We have then considered 3 matching problems, also involving real-world examples. Let us first discuss matching results from our running example, see,

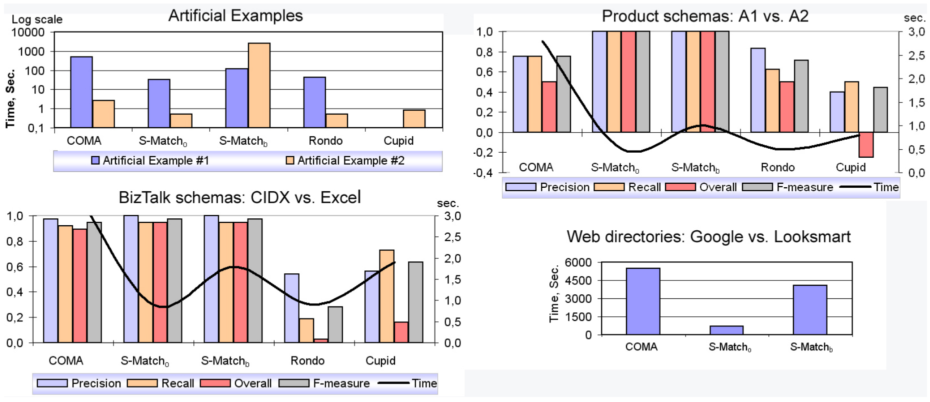

Fig. 2: Evaluation Results

Figure 2 (Product schemas: A1 vs. A2). There, S-Match outperforms the other systems in terms of quality indicators. Since all the labels at nodes in the given test case were correctly encoded into propositional formulas, all the quality measures of S-Match reach their highest values. In fact, as discussed before, the propositional SAT solver is correct and complete. This means that once the element level matchers have found all and only the mappings, S-Match will return all of them and only the correct ones. Also, S-Match$_o$ works more than 5 times faster than COMA, 1.5 times faster than Cupid, and as fast as Rondo.

For a pair of BizTalk schemas: CIDX vs. Excel, S-Match performs as good as COMA and outperforms the other systems in terms of quality indicators. Also, S-Match$_o$ works more than 4 times faster than COMA, more than 2 times faster than Cupid, and as fast as Rondo.

For the biggest matching problem (Web Directories: Google vs. Looksmart), which contains hundreds and thousands of nodes, unfortunately, we did not have enough human resources to create expert mappings for this test case (we are still working on establishing them), and thus, for the moment we have evaluated only the performance *time*. S-Match$_o$ performs about 9 times faster than COMA, and about 7 times faster than S-Match$_b$. Rondo and Cupid run out of memory, therefore we do not report any results for them.

6.2 Evaluation Summary

Quality measures. Since most matching systems return similarity coefficients, rather than semantic relations, our qualitative analysis was based on the measures developed for those systems. Therefore, we had to omit information about the type of relations S-Match returns, and focus only on the number of present/absent mappings. We totally discarded from our considerations the disjointness relation, however, its value should not be underestimated, because this relation reduces the search space. For the example of Figure 1, if Cupid would support the analysis of dissimilarity between schema elements, it could possibly recognize that $C1_4$ is disjoint (dissimilar) with $C2_4$, and then avoid false positives such as determining that $C1_{10}$ is similar to $C2_{11}$, and so on.

Pre-match efforts. Typically, these efforts include creating a precompiled thesaurus with relations among common and domain specific terms. On the one side, such a thesaurus can be further reused, since many schemas to be matched are similar to already matched schemas, especially if they are describing the same application domain. On the other side, for the first schemas to be matched from a novel domain, creation of such a thesaurus requires time. With this respect, exploiting an external resource of common and domain knowledge (e.g., WordNet) can significantly reduce the pre-match efforts. In the example of Figure 1, in order for Cupid to determine $C1_7$ as an appropriate match for $C2_8$, we have to add an entry <Hyp key="brand:name"> 0.7</Hyp> to its thesaurus, while S-Match obtains the knowledge of the hyponymy relation in the above case automatically from WordNet.
Performance measures. *Time* is a very important indicator, because when matching industrial-size schemas (e.g., with hundreds and thousands of nodes, which is quite typical for e-business applications), it shows scalability properties of the matchers and their potential to become an industrial-strength systems. It is also important in web applications, where some weak form of real time performance is required (to avoid having a user waiting too long for the system respond).

7 Conclusions

We have presented a new semantic schema matching algorithm and its optimizations. Our solution builds on the top of the past approaches at the element level and introduces a novel (with respect to schema matching) techniques, namely model-based techniques, at the structure level. We conducted a comparative evaluation of our approach implemented in the S-Match system against three state of the art systems. The results empirically prove the strengths of our approach.

Future work includes development of the *iterative* and *interactive* semantic matching system. It will improve the quality of the mappings by iterating and by focusing user's attention on the critical points where his/her input is maximally useful. S-Match works in a top-down manner, and hence, mismatches among the top level elements of schemas can imply further mismatches between their descendants. Therefore, next steps include development of a *robust* semantic matching algorithm. Finally, we are going to develop a *testing methodology* which is able to estimate quality of the mappings between schemas with hundreds and thousands of nodes. Initial steps have already been done, see for details [1]. Here, the key issue is that in these cases, specifying expert mappings manually is (often) neither desirable nor feasible task. Comparison of matching algorithms on real-world schemas from different application domains will also be performed more *extensively.*

References

1. P. Avesani, F. Giunchiglia, and M. Yatskevich. A large scale taxonomy mapping evaluation. In *Proceedings of ISWC*, 2005.

2. S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Record*, pages 54–59, 1999.
3. P. Bernstein, S. Melnik, M. Petropoulos, and C. Quix. Industrial-strength schema matching. *SIGMOD Record*, 33(4):38 – 43, 2004.
4. D. Le Berre. A satisfiability library for Java. http://www.sat4j.org/.
5. P. Bouquet, L. Serafini, and S. Zanobini. Semantic coordination: A new approach and an application. In *Proceedings of ISWC*, pages 130–145, 2003.
6. M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, (5(7)):394–397, 1962.
7. R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: Discovering complex semantic matches between database schemas. In *Proceedings of SIGMOD*, pages 383–394, 2004.
8. H. H. Do and E. Rahm. COMA - a system for flexible combination of schema matching approaches. In *Proceedings of VLDB*, pages 610–621, 2002.
9. J. Euzenat and P. Valtchev. Similarity-based ontology alignment in OWL-lite. In *Proceedings of ECAI*, pages 333–337, 2004.
10. A. Gal, A. Anaby-Tavor, A. Trombetta, and D. Montesi. A framework for modeling and evaluating automatic semantic reconciliation. *VLDB Journal*, (14(1)), 2005.
11. F. Giunchiglia and P. Shvaiko. Semantic matching. *KER Journal*, (18(3)), 2003.
12. F. Giunchiglia, P. Shvaiko, and M. Yatskevich. S-Match: an algorithm and an implementation of semantic matching. In *Proceedings of ESWS*, pages 61–75, 2004.
13. F. Giunchiglia, M. Yatskevich, and E. Giunchiglia. Efficient semantic matching. In *Proceedings of ESWC*, pages 272–289, 2005.
14. N. Guarino. The role of ontologies for the Semantic Web (and beyond). Technical report, Laboratory for Applied Ontology, ISTC-CNR, 2004.
15. V. Haarslev, R. Moller, and M. Wessel. RACER: Semantic middleware for industrial projects based on RDF/OWL, a W3C Standard. `http://www.sts.tu-harburg.de/~r.f.moeller/racer/`.
16. J. Kang and J. F. Naughton. On schema matching with opaque column names and data values. In *Proceedings of SIGMOD*, pages 205–216, 2003.
17. J. Madhavan, P. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proceedings of VLDB*, pages 49–58, 2001.
18. B. Magnini, L. Serafini, and M. Speranza. Making explicit the semantics hidden in schema models. In *Proceedings of workshop on HLTSWWS at ISWC*, 2003.
19. S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm. In *Proceedings of ICDE*, pages 117–128, 2002.
20. S. Melnik, E. Rahm, and P. Bernstein. Rondo: A programming platform for generic model management. In *Proceedings of SIGMOD*, pages 193–204, 2003.
21. A. G. Miller. WordNet: A lexical database for English. *Communications of the ACM*, (38(11)):39–41, 1995.
22. J. Z. Pan. *Description Logics: reasoning support for the Semantic Web*. PhD thesis, School of Computer Science, The University of Manchester, 2004.
23. D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, (2):293–304, 1986.
24. E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, (10(4)):334–350, 2001.
25. P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics*, IV, 2005.
26. M. K. Smith, C. Welty, and D. L. McGuinness. OWL web ontology language guide. Technical report, World Wide Web Consortium (W3C), http://www.w3.org/TR/2004/REC-owl-guide-20040210/, February 10 2004.